LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Id: a language with implicit parallelism

Id solutions to the four "Salishan problems"

Computation Structures Group Memo 305
February 8, 1990

Rishiyur S. Nikhil
Arvind

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Preface

In the 1988 High-Speed Computer Conference at Gleneden, Oregon, the *Salishan Conference*, programs for four applications were studied in a variety of parallel languages. Based on this, a book is being assembled, with each chapter showing the solutions in a different programming language. Each chapter is written by an expert in that language. The book is to be published by North Holland.

This CSG memo contains the Id chapter for the book. The four problem statements are given in the appendix.

Id: a language with implicit parallelism

Rishiyur S. Nikhil *and* Arvind

Massachusetts Institute of Technology

Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA
e-mail: nikhil@lcs.mit.edu *and* arvind@lcs.mit.edu

1 Introduction

Id is a parallel programming language developed in the Computation Structures Group at MIT's Laboratory for Computer Science.[1] In developing Id, we have three major goals.

High level: at least as expressive as modern functional languages and Lisp. Parallelism in Id is implicit— the user does not have to manage partitioning, scheduling and synchronization.

General purpose: suitable for both "scientific" and "symbolic" computation. Id has efficient arrays and floating-point operations, as well as recursive data structures (*e.g.*, lists) in an automatically managed heap.

High performance: Our aim is for an Id program compiled for a dataflow machine to achieve at least as much absolute performance as its FORTRAN counterpart on a von Neumann machine built with comparable technology.[2]

Id is a layered language [1, 2], with layer 0 representing the cleanest semantics and layer 2 representing the most expressive power. Layer 0 is purely functional, and is similar to other modern functional languages like Miranda and Haskell, *i.e.*, it has higher-order functions, non-strict semantics, polymorphic types with static type-checking by inference, algebraic types with pattern-matching, list comprehensions, and user-defined abstract data types. Id's "array comprehensions" are fairly unique.

Layer 1 adds "I-structures" to layer 0. These permit a limited form of assignment. One can allocate data structures with empty slots, assign values to these slots, and read values from these slots. A slot can be assigned a value no more than once. Reading a value from a slot is automatically blocked until it has been assigned a value. This addition sacrifices the referential transparency of layer 0, but retains

[1] A first version of Id appeared in 1978. Since 1985, it underwent a series of revisions during which it was variously called Id/83s, Id86, Id Noveau, *etc.*, finally reverting to just "Id" again.

[2] However, Id is in no way specific to dataflow machines.

determinacy, since the value read from a slot does not depend on the time that the program attempts to read it. The loss of referential transparency implies a certain loss in the ability to transform programs (*e.g.*, for proving correctness, for program optimization, *etc.*); however, the benefits are:

- Certain programs that involve excessive copying when written functionally can now be written more efficiently.
- Certain programs that must be written recursively when written functionally can now be written using loops, *i.e.*, tail-recursively.

Most of the programs in this chapter do not use I-structures explicitly (a small use is made in the Doctor's office section). However, the Id compiler uses I-structures extensively to implement all data structures, including those from layer 0.

Layers 0 and 1 are purely determinate, *i.e.*, answers depend only on inputs. Layer 2 introduces non-determinism by adding "Managers" to layer 1, for those applications that need it, such as shared-resource problems and operating systems. An example will be seen in the Doctor's office section.

It is possible to instruct the Id compiler to only accept programs restricted to layer 0 or layer 1, since each layer involves new syntactic constructs.

1.1 A brief introduction to the language

Id programs are built up from *expressions*. In addition to standard infix operators like "+", Id uses juxtaposition to indicate function application:

```
f e1 ... en
```

Functions are defined using `def`:

```
def clip top y = if (y > top) then top else y ;
```

Functions are curried, and application associates to the left, so that the expression:

```
clip 5
```

denotes a function of one argument that clips its argument to 5.

Data structures in Id are defined using *algebraic types*. However, some data structures are so useful that they are pre-defined with special notation. An *n-tuple* may be constructed by listing n expressions separated by commas. Here is a 2-tuple:

```
(a+b),(a-b)
```

A list is either empty (`Nil`), or constructed using the infix "cons" operator ($e_1 : e_2$). Destructuring of lists (testing for emptiness, acessing the head and tail) is usually done using pattern-matching— function on lists are defined in several clauses:

```
def length Nil     = 0                          % for empty lists
 | length (x:xs) = 1 + length xs ;              % for non-empty lists
```

The clauses must have disjoint patterns. The second clause binds `x` and `xs` to the head and tail, respectively, of the non-empty argument list.

A local scope may be created using a *block*. Blocks may be nested, with standard lexical scoping. The bindings in a block can include function definitions:

```
% integrate function f(x) from a to b
def integrate f a b = { delta = 0.0001 ;
                        def iter x s = if x > b then
                                         s
                                       else
                                         iter (x + delta) (s + f x) ;
                    In
                      delta * (iter a 0) } ;
```

The block contains one ordinary binding and one function binding. The value returned by the block (and the `integrate` function) is the value of the expression following the `In` keyword. Note that `integrate` is higher-order— its first argument is itself a function.

Although recursion subsumes iteration, Id also has `while`-and `for`-loop constructs.

Like other functional languages, Id also has *list comprehensions*. The following expression creates all pairs (x, y) such that $y \leq x$ and $x^2 + y^2 \leq 25$ (all pixels in first octant within radius 5):

```
{: (x,y) || x <- 0 to 5 & y <- 0 to x when x*x + y*y <= 25 }
```

Terms like "`x <- 0 to 5`" are called *generators* and terms like "`when ...`" are called *filters*. Generators and filters are scoped from left to right, *i.e.*, they can use identifiers bound in generators to their left.

Id also has arrays, which are constructed using *array comprehensions*. This expression denotes an identity matrix of size $n \times n$:

```
{matrix (1,n),(1,n)
| [i,j] = 0 || i <- 2 to n & j <- 1 to (i-1)      % below diagonal
| [i,i] = 1 || i <- 1 to n                        % diagonal
| [i,j] = 0 || j <- 2 to n & i <- 1 to (j-1) }    % above diagonal
```

In general, arrays can have arbitrary lower and upper bounds, and they can be queried at run time using the `bounds` function. The array contents in the example are specified in three regions— below, on and above the diagonal, respectively. The specifications must be disjoint— a run time error will catch multiple definitions. The generator syntax is identical to that in list comprehensions.

For efficiency reasons, we distinguish vectors, matrices, 3-dimensional arrays, *etc.* on the basis of type.

1.2 Non-strict, but not lazy

Lazy and eager evaluation are *not* synonymous with strict and non-strict semantics, respectively (even though this confusion is widespread in the literature). The former terms are concerned with operational semantics (what interpreters do), while the latter terms are concerned with denotational semantics (declarative meanings of programs). Non-strictness may be achieved by both lazy as well as eager (parallel) evaluators.

When a lazy evaluator encounters an application `f(arg)`, *no* computational resources are devoted to `arg`. Instead, `arg` is packaged into a closure and passed to `f`. If `f` ever requires the value of `arg`, the evaluator then devotes all its computational resources to it, *i.e.*, it invokes the closure.

On the other hand, Id achieves non-strictness with a parallel, eager evaluator. Computational resources are shared amongst `f` and `arg`, *i.e.*, `f` is invoked in parallel with `arg`, passing only a place-holder for `arg` to `f`. If `f` ever requires the value of `arg`, it blocks on the place-holder. If `f` ignores its argument, it simply discards the place-holder. Eager evaluation is thus speculative.

However, not all computation in Id is speculative. In particular, the arms of conditionals (more-generally, `case` expressions) are not evaluated speculatively. After the predicate has determined which arm is required, only that arm is evaluated. This is how we control recursion, and our experience has been that with this control, in almost all cases, the potential waste of resources due to speculative argument evaluation is not a problem.

The advantages of Id's eager evaluation are that (a) we avoid the overhead of building a closure for `arg` and later invoking it, and (b) the computation of `arg` is begun before it is really demanded, thus increasing the parallelism and shortening the critical path. Consider the following Id expression representing an array of the first 20 Fibonacci numbers:

```
{ fibs = {array (1,20)
         | [i] = 1                          || i <- 1 to 2
         | [i] = fibs[i-1]+fibs[i-2] || i <- 3 to 20}
In
  fibs }
```

Such a recursive array definition is only possible in non-strict languages like Id, Haskell, and Miranda, and is not possible in strict languages like Lisp and ML. However, in this example, there is no need at all for lazy evaluation. In fact, the overhead of building closures for all the array components is likely to far outstrip the cost of computing the array in the first place.

The disadvantage of eager evaluation is that if a value is never demanded, the resources allocated to compute it are wasted. The extreme case of this occurs when the value represents an infinite structure. We handle these cases by special annotations that request lazy evaluation. These will be described in our solution to the Hamming problem.

An approach currenly being investigated by various researchers is to start with lazy evaluation as the default, and to use *strictness analysis* to predict where it is safe to evaluate things eagerly. However, it is too early to judge its effectiveness on large programs with data structures and higher-order functions. Further, it will have no effect on our `fibs` program above, which *requires* non-strictness. It appears that in lazy languages, it will be necessary to have annotations to suggest "eager" evaluation.

We note in passing that the Haskell language definition only requires non-strict semantics— it takes no position on lazy or eager evaluation. This is appropriate, for it leaves implementors with some latitude for experimentation.

1.3 Implementations of Id

Our efforts have concentrated on compilation of Id for dataflow machines which, to date, have been emulated in software. A complete programming environment for Id, called "Id World", is available under license from MIT for a small fee. It contains a compiler that translates Id programs into the machine code for the MIT Tagged-Token Dataflow Architecture (TTDA). In addition, Id World contains GITA, an emulator for the TTDA. Extensive instrumentation in GITA permits the experimenter to collect and plot various statistics such as parallelism profiles, instruction counts, instruction mixes, resource usage profiles, *etc.*

We are currently building a real dataflow machine called Monsoon. An early single-processor prototype of Monsoon has been running compiled Id since October 1988. In collaboration with Motorola, we are building new, multi-processor Monsoon machines which are expected to be available in the summer of 1989. We aim to retain the current Id World interface for Monsoon, so that programs can be developed today for Monsoon.

We have recently begun to study compilation of Id for other sequential and parallel machines as well, which will increase its availability and value to other researchers.

1.4 Our test runs: parallelism profiles, instruction counts and critical path lengths

All the Id programs in this chapter were run on GITA, and we present their parallelism profiles, instruction counts, and critical path lengths. The emulator was run in "idealized" mode, *i.e.*, with the following assumptions:

- All instructions take one time unit to execute.
- Each instruction executes as soon as its input data are ready (*i.e.*, immediately after all its predecessor instructions in the dataflow graph have executed).
- It takes zero time to communicate data from an instruction to its successor in the dataflow graph.

The *parallelism profile* is a plot of the number of instructions executed at each time step. The *total instruction count*, therefore, is the area under the curve. The *critical path length* is that time step after which no more instructions execute.

While this "idealized" mode is admittedly unrealistic, it is very useful in showing what is the maximum parallelism available under some algorithm. As some of the problems in this chapter demonstrate, the parallelism of some algorithms is not at all obvious.

2 Hamming's problem, extended

Our program is shown in Figure 1, with `hamming_ext` as the top-level function.

In order to explain the solution, we begin with a solution for the original (simpler) Hamming problem, where the primes are limited to 2, 3 and 5. It is a direct implementation of the observation that if h is in the result hs, then $2h$, $3h$ and $5h$ are also in hs:

```
def hamming n =
    { hs = 1 : merge_2 (mapmult 2 hs)
                       (merge_2 (mapmult 3 hs)
                                (mapmult 5 hs)) ;
      In
        until_n n hs } ;
```

Here, `hs` is a stream, *i.e.*, a potentially infinite list. `Mapmult` takes an integer p and a stream $x_1, x_2, \ldots$ and produces the stream $px_1, px_2, \ldots$. `Merge_2`, takes two streams in ascending order and merges them into a new stream in ascending order, removing duplicates. `Until_n` produces the prefix of a stream containing just those numbers that are $\leq n$. These functions are shown in Figure 1

```
def hamming_ext primes n =
    { def f xs p = {  h = merge_2 (mapmult p (1:h)) xs
                   In
                      h } ;
      hs = 1: foldl_list f Nil primes ;
    In
      until_n n hs } ;

def mapmult p Nil     = Nil
 |  mapmult p (x:xs) = (p*x):#(mapmult p xs);

def merge_2 Nil    Nil    = Nil
 |  merge_2 (x:xs) Nil    = (x:xs)
 |  merge_2 Nil    (y:ys) = (y:ys)
 |  merge_2 (x:xs) (y:ys) = if (x < y) then
                              x :# merge_2 xs (y:ys)
                            else if (x > y) then
                              y :# merge_2 (x:xs) ys
                            else
                              x :# merge_2 xs ys ;

def until_n n Nil    = Nil
 |  until_n n (x:xs) = if (x <= n) then
                         x:(until_n n xs)
                       else
                         Nil ;
```

Figure 1: Id program for the extended Hamming problem.

In each of these functions, we are dealing with potentially infinite lists. Thus, we use the "lazy-tail" list-constructor ":#" to override Id's default eager evaluation. It is also possible to delay the head, or both the head and the tail, by using the constructors "#:" and "#:#", respectively. The annotations are only in the constructor—component selection is identical for delayed and non-delayed components.

The above program is inefficient because the three streams that are merged together contain several duplicates (*e.g.*, 2*3, 3*2) that are then removed during the merge. Here is a new solution that avoids building duplicates in the first place:

```
def hamming n =
    { hs = 1: {  s1 = mapmult 2 (1:s1) ;
                 s2 = merge_2 (mapmult 3 (1:s2)) s1 ;
                 s3 = merge_2 (mapmult 5 (1:s3)) s2 ;
              In
                s3 } ;
    In
      until_n n hs } ;
```

Here, `s1` contains all the powers of 2, `s2` merges in all products with all powers of 3, and so on.

Finally, we can generalize our last solution so that, instead of working with just the primes 2, 3, and 5, it works with a list of primes. This is shown in the function `hamming_ext` in Figure 1. The function `foldl_list` performs the nested merge of our previous solution, *i.e.*,

```
foldl_list f Nil (p1:...:pN:Nil) ⇒ (f ( ... (f (f Nil p1) p2) ... ) pN)
```

It is available as a library function in Id, but it can also be defined as:

```
def foldl_list f z Nil     = z
 | foldl_list f z (x:xs) = foldl_list f (f z x) xs ;
```

2.1 A test run

We ran the following program on GITA, our dataflow emulator:

```
hamming_ext (3:5:7:11:13:17:19:23:Nil) 5000
```

with the following output:

```
(1 3 5 7 9 11 13 15 17 19 21 23 25 27 33 35 39 45 49 51 55 57 63 65 69
75 77 81 85 91 95 99 105 115 117 119 121 125 133 135 143 147 153 161
165 169 171 175 187 189 195 207 209 221 225 231 243 245 247 253 255
```

```
273 275 285 289 297 299 315 323 325 343 345 351 357 361 363 375 385
391 399 405 425 429 437 441 455 459 475 483 495 507 513 525 529 539
561 567 575 585 595 605 621 625 627 637 663 665 675 693 715 729 735
741 759 765 805 819 825 833 845 847 855 867 875 891 897 931 935 945
969 975 1001 1029 1035 1045 1053 1071 1083 1089 1105 1125 1127 1155
1173 1183 1197 1215 1225 1235 1265 1275 1287 1309 1311 1323 1331 1365
1375 1377 1425 1445 1449 1463 1485 1495 1521 1539 1547 1573 1575 1587
1615 1617 1625 1683 1701 1715 1725 1729 1755 1771 1785 1805 1815 1859
1863 1875 1881 1911 1925 1955 1989 1995 2023 2025 2057 2079 2093 2125
2145 2185 2187 2197 2205 2223 2261 2275 2277 2295 2299 2375 2401 2415
2431 2457 2475 2499 2527 2535 2541 2565 2601 2625 2645 2673 2691 2695
2717 2737 2783 2793 2805 2835 2873 2875 2907 2925 2975 3003 3025 3059
3087 3105 3125 3135 3159 3179 3185 3211 3213 3249 3267 3289 3315 3325
3375 3381 3465 3519 3549 3553 3575 3591 3645 3675 3703 3705 3757 3773
3795 3825 3861 3887 3927 3933 3969 3971 3993 4025 4095 4125 4131 4165
4199 4225 4235 4275 4301 4335 4347 4375 4389 4455 4459 4485 4563 4617
4641 4655 4675 4693 4719 4725 4761 4807 4845 4851 4875 4913)
```

which is not in Id syntax because GITA, written in Lisp, simply prints out the Lisp value of the result. The parallelism profile generated is shown in Figure 2.

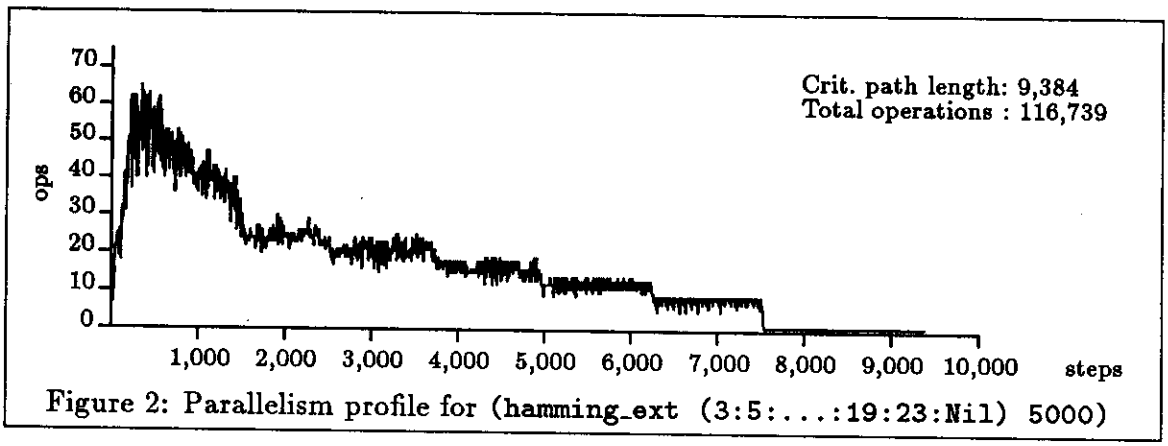

Figure 2: Parallelism profile for `(hamming_ext (3:5:...:19:23:Nil) 5000)`

3 The paraffins problem

Turner's original solution [9] was written in the language KRC. It can be transcribed practically verbatim into Id, since the functional core of Id is similar to KRC (including list comprehensions), and Id shares the same non-strict semantics as KRC. However, that solution is quite inefficient, because it generates many duplicates only to be filtered out later. In [3], we showed an efficient program that avoids generating duplicates in the first place, using the canonical tree-enumeration techniques

described in [4] (and discovered independently by S.K. Heller, our co-author in [3]). That solution is repeated here, and is shown in two parts: Figure 3 shows the code for the sub-problem of generating radicals, and Figure 4 shows the generation of paraffins, with top-level function `paraffins_until`. For more details, including a discussion of the development of the solution, please see [3].

```
type radical = H | C radical radical radical ;

def 3_partitions m =
        {: (i,j,k) || i <- 0 to floor (m/3)
                   & j <- i to floor ((m-i)/2)
                   & k  =  m - (i + j) } ;

def remainders Nil     = Nil
 |  remainders (r:rs) = (r:rs) : (remainders rs) ;

def radical_generator n =
    { radicals = {array (0,n)
                 | [0] = H:nil
                 | [j] = rads_of_size_n radicals j || j <- 1 to n}
    In
      radicals} ;

def rads_of_size_n radicals n =
    {: C ri rj rk || (i,j,k) <- 3_partitions (n-1)
                  & ri:ris <- remainders (radicals[i])
                  & rj:rjs <- remainders (if (i == j) then ri:ris
                                                      else radicals[j])
                  & rk     <- if (j == k) then rj:rjs
                              else radicals[k] } ;
```

Figure 3: Id program for the generating radicals for the paraffins problem.

3.1 Radicals

A radical is a paraffin with a single hydrogen atom removed, *i.e.*, a molecule with formula C_iH_{2i+1}. The structure of such molecules can be recursively described as either

- a Hydrogen atom, or
- a Carbon atom attached to three other radicals

This is expressed in `radical` type-declaration in Figure 3. It consists of two disjuncts with constructors `H` and `C`, respectively. In the latter case, there are four components,

each of which is itself of type `radical`.

By way of illustration, here are some more examples of algebraic type declarations. The type of booleans can be declared:

```
type bool = False | True ;
```

The type of binary trees with integers in the nodes can be declared:

```
type tree = Leaf | Node int tree tree ;
```

And, the type of lists can be declared:

```
type (list *0) = Nil | Cons *0 (list *0) ;
```

The list type is *polymorphic* because it is parameterized by a type variable `*0`, so that we can have lists of integers, lists of booleans, lists of lists of integer-to-integer functions, *etc.*

3.2 Generating radicals

Suppose we wish to generate all radicals of size n. For $n > 0$, the radical will have one carbon as its "root" carbon, and three sub-radicals of collective size $n-1$. Thus, we need to partition $n-1$ into three sizes in order to generate the sub-radicals. However, if we use all possible 3-partitions of $n-1$, we will generate many duplicate radicals because the partition (i,j,k) is equivalent to (i,k,j), (j,i,k), and so on. We can avoid this by insisting that $i \leq j \leq k$. The function `3_partitions` is a function to generate a list of all three partitions of m $(= n-1)$ in this canonical order.

The generation of radicals of size n can defined recursively. When $n = 0$, there is only one such radical— a lone hydrogen atom. When $n > 0$, we construct all canonical 3-partitions (i,j,k) of $n-1$; for each such partition, we generate, recursively, all radicals `ri` of size i, all radicals `rj` of size j and all radicals `rk` of size k, and constuct the new radical `(C ri rj rk)`. Here is the function:

```
def rads_of_size_n n =
    if (n == 0) then
      H:Nil
    else
      {: C ri rj rk || (i,j,k) <- 3_partitions (n-1)
                    & ri <- rads_of_size_n i
                    & rj <- rads_of_size_n j when (le? ri rj)
                    & rk <- rads_of_size_n k when (le? rj rk) } ;
```

where `le?` is some function that checks that its two radical arguments are in canonical order. The first `when` clause takes care of the following situation: when $i = j$, since `ri`

and `rj` range over all possible pairs of radicals of size i, they may not be in canonical order— the `when` clause filters out these pairs. Similarly, the second `when` clause filters out duplicates when $j = k$.

We can avoid this generation-of-duplicates-and-filtering as follows. For each i, let `ri` range over $ri_1, ri_2, \ldots$. Then, when $i = j$ and, when `ri` is, say, ri_4, we make `r2` range over $ri_4, ri_5, \ldots$. Thus, for each element of the list, we would like to have access not only to that element, but also to the remainder of that list.

In particular, we need a function that, given $ri_1, ri_2, ri_3, \ldots$, produces the list of lists:

$$(ri_1, ri_2, ri_3, ri_4, ri_5, \ldots), (ri_2, ri_3, ri_4, ri_5, \ldots), (ri_3, ri_4, ri_5, \ldots), \ldots$$

The function that performs this is called `remainders` (Figure 3).

Now, we can improve our `rads_of_size_n` function:

```
def rads_of_size_n n =
    {: C ri rj rk || (i,j,k) <- 3_partitions (n-1)
                  & ri:ris <- remainders (rads_of_size_n i)
                  & rj:rjs <- remainders (if (i == j) then ri:ris
                                          else rads_of_size_n j)
                  & rk     <- if (j == k) then rj:rjs
                              else rads_of_size_n k } ;
```

However, there is still a major inefficiency in this function. It has the classical Fibonacci recursion, because to compute radicals of, say, size 3, we compute radicals of size 0, 1, and 2, but to compute radicals of size 2, we compute radicals of size 0 and 1, and so on. In other words, we recompute radicals of each size too often.

We use a standard trick— use an array to cache, at index n, the list of radicals of size n, and just look up this array each time we need radicals of size n. In Figure 3, `radical_generator` is the function that constructs this array, using `rads_of_size_n` to compute each component. And, `rads_of_size_n` itself uses this array to find radicals of size $< n$.

3.3 Paraffins from radicals

Or, Molotov cocktails?

Please refer to Figure 4.

In order to define a canonical form for paraffins, we observe that every paraffin of size n has either

```
type paraffin = BCP radical radical | CCP radical radical radical radical ;

def BCP_generator radicals n =
        if (odd? n) then
          Nil
        else
          {: BCP r1 r2 || r1:r1s <- remainders (radicals[floor (n/2)])
                       & r2 <- r1:r1s } ;

def 4_partitions m =
        {: (i,j,k,l) || i <- 0  to floor (m/4)
                     & j <- i to floor ((m-i)/3)
                     & k <-    (max j (ceiling (m/2-i-j)))
                           to (floor ((m-i-j)/2))
                     & l =  m - (i+j+k) } ;

def CCP_generator radicals n =
        {: CCP ri rj rk rl || (i,j,k,l) <- 4_partitions (n-1)
                           & ri:ris <- remainders (radicals[i])
                           & rj:rjs <- remainders (if i==j then ri:ris
                                                   else radicals[j])
                           & rk:rks <- remainders (if j==k then rj:rjs
                                                   else radicals[k])
                           & rl <- if (k==l) then rk:rks
                                   else radicals[l] } ;

def paraffins_until n =
    { radicals = radical_generator (floor (n/2)) ;
    In
      {array (1,n)
      | [j] = (BCP_generator radicals j),
              (CCP_generator radicals j)   || j <- 1 to n}} ;
```

Figure 4: Id program for the paraffins problem.

- a unique *bond center*, *i.e.*, a bond with $\frac{n}{2}$-sized radicals on its two sides, or
- a unique *carbon center*, *i.e.*, a carbon attached to 4 radicals, each of size $< \frac{n}{2}$.

This is expressed in the `paraffin` type-declaration in Figure 4. We can then define our canonical representation as either `(BCP r1 r2)` where `r1` $\leq$ `r2`, or `(CCP r1 r2 r3 r4)` where `r1` $\leq$ `r2` $\leq$ `r3` $\leq$ `r4`.

Bond-centered paraffins can be enumerated using the function `BCP_generator`, whose first argument should be the array of radicals defined in the previous section, *i.e.*, from the list of radicals of size $\frac{n}{2}$, we draw all pairs `r1` and `r2` such that `r2` does not precede `r1` in the list.

To enumerate carbon-centered paraffins in canonical order, we follow a strategy similar to the one used to produce radicals. For a paraffin of size n, one carbon is the center, so we find all canonical 4-partitions (i, j, k, l) of $n - 1$, representing the sizes of the four attached radicals (*i.e.*, $i \leq j \leq k \leq l$). For each of these 4-partitions, we take all the radicals `ri`, `rj`, `rk` and `rl` of sizes i, j, k, and l, respectively. We use the same "remainders" trick to take care of avoiding duplicates when $i = j$, $j = k$ and $k = l$. Using these radicals, we form the paraffins `(CCP r1 r2 r3 r4)`.

Finally, our top-level function `paraffins_until` takes a numeric argument n and generates an array of size n such that the n'th index contains a pair of lists— a list of all bond-centered paraffins of size n and a list of all carbon-centered paraffins of size n. Each list is in canonical order. It would be easy to flatten this into a single list, if desired.

3.4 A test run

The parallelism profile for `(paraffins_until 15)` is shown in Figure 5. The number of paraffins containing n carbons, for $n = 1, 2, \cdots, 15$ are 1, 1, 1, 2, 3, 5, 9, 18, 35, 75, 159, 355, 802, 1858 and 4347, respectively. The parallelism of this program is not at all obvious from the algorithm.

4 A doctor's office

It is not clear which of the following programs is requested in the problem statement.

- A *determinate* simulation in which the non-determinism of the real world is modelled by an oracle that is a parameter to the program; or
- A program that is itself non-deterministic.

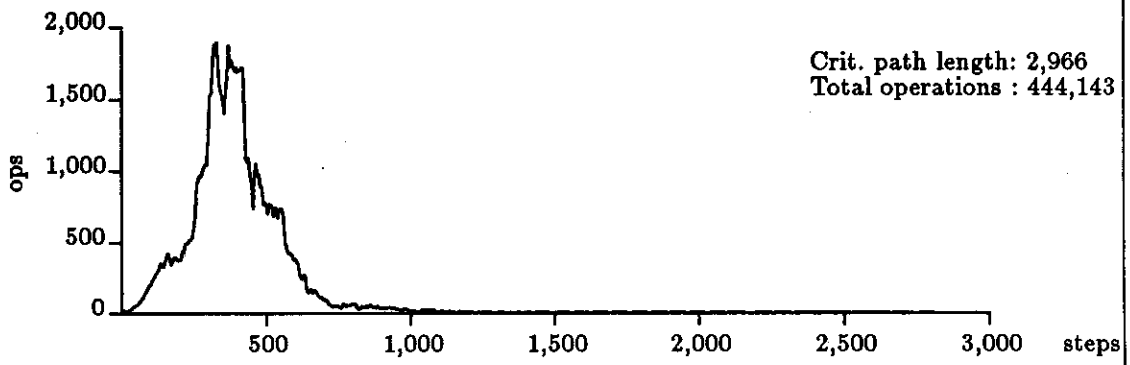

Figure 5: Parallelism profile for (paraffins_until 15)

If the intent is really to simulate a doctor's office, then the former program makes more sense, because it is repeatable and we can control the choices made by the oracle. However, if the intent is to see how non-determinism is handled by the programming language (for example to evaluate its suitability for operating systems code), then the latter program makes more sense. Accordingly, we have developed solutions for both interpretations.

We begin by discussing how (pseudo-)random numbers may be generated and used in Id.

4.1 Random numbers

Here is a function for the linear congruence method of generating random numbers[3]. Given a seed `X`, it returns a random number `r` in the range 0 to 1 and a new seed `X'`.

```
def rand_fn X =  { a = 25173 ; c = 13849 ; m = 65536 ;
                   r = X / (m - 1) ;
                   X' = mod (a*X + c) m ;
                 In
                   r,X' } ;
```

Streams of random numbers: We can use this function, for example, to produce a stream of random numbers, given an initial seed `X0`:

```
def mk_random_stream X0 = { def mk_rs X = { r,X' = rand_fn X
                                          In
                                            r:# mk_rs X'} ;
                          In
                            mk_rs X0 } ;
```

[3]See [5] for an extensive discussion.

Here, we have used the "lazy-tail" list constructor ":#" to delay the construction of the tail of the stream, since we are likely to use only a small prefix. For more efficiency, it is possible to mix regular and lazy list constructors so that, for example, the stream is delayed at every 100'th element.

Picking a random element of a list: Given a random number `r` in the range 0 to 1, we can use the following function to pick a random member of a list. `Pick_random` not only returns the random element, but also the other members of the list:

```
def pick_random r xs =
    { n = round (r * (length xs - 1)) ;
      def separate j (x:xs) = if (j == n) then
                                x,xs
                              else
                                { x',xs' = separate (j+1) xs
                                In
                                  x',(x:xs') }
    In
      separate 0 xs } ;
```

Picking a random event: Suppose we want to choose one of three possible events with probabilities 1%, 54% and 45% respectively. Here is a function we can use:

```
def choose_event r = if (r <= 0.01) then 0
                     else if (r <= 0.55) then 1
                     else 2 ;
```

where, again, `r` is a random number in the range 0 to 1.

4.2 A determinate simulator for the doctor's office

Our determinate solution is shown in Figure 6.

The state of the system at any time can be modelled by seven items:

- `wellps`, a collection of well patients.
- `waitps`, a queue of sick patients waiting for doctors.
- `waitds`, a queue of doctors waiting for sick patients.
- `consults`, a collection of patient-doctor pairs (in consultation).
- Three "histories": `hist1`, a list of patients as they fall sick, `hist2`, a list of patient-doctor pairs as they go into consultation, and `hist3`, a list of patients as they get cured.

There are only two kinds of events that drive the system:

```
def sick_event r (wellps, consults, waitps, waitds, hist1, hist2, hist3) =
    if (wellps == Nil) then
      (wellps, consults, waitps, waitds, hist1, hist2, hist3)
    else
      { (p,wellps') = pick_random r wellps
      In
        if (waitds == Nil) then
          (wellps', consults, (waitps ++ p:Nil), waitds, p:hist1, hist2, hist3)
        else
          { (d:waitds') = waitds
          In
            (wellps', (p,d):consults, waitps, waitds',
             p:hist1, (p,d):hist2, hist3)}} ;

def cure_event r (wellps, consults, waitps, waitds, hist1, hist2, hist3) =
    if (consults == Nil) then
       (wellps, consults, waitps, waitds, hist1, hist2, hist3)
    else
       { (p,d),consults' = pick_random r consults
       In
         if (waitps == Nil) then
           (p:wellps, consults', waitps, waitds ++ d:Nil, hist1, hist2, p:hist3)
         else
           { (p':waitps') = waitps
           In
             (p:wellps, (p',d):consults, waitps', waitds,
              hist1, (p',d):hist2, p:hist3) }} ;

def process randoms (wellps, consults, waitps, waitds, hist1, hist2, hist3) =
    {  r1:r2:randoms' = randoms ;
       e = choose_event r1
    In
       {case e of
          0 = reverse hist1, reverse hist2, reverse hist3
       |  1 = process randoms'
                      (sick_event r2
                                  (wellps, consults, waitps, waitds,
                                   hist1, hist2, hist3))
       |  2 = process randoms'
                      (cure_event r2
                                  (wellps, consults, waitps, waitds,
                                   hist1, hist2, hist3)) } };

def simulate randoms patients doctors =
    process randoms
            (patients, Nil, Nil, doctors, Nil, Nil, Nil) ;
```

Figure 6: Determinate program for the doctor's office problem.

- A `sick_event`: Some well patient `p` falls sick. Of course, this can only happen if `wellps` is non-empty.

  Effects: We use `r`, a random number in the range 0 to 1 to choose which patient falls sick. If a doctor `d` is available (`waitds` is non-empty), `p` and `d` go into consultation (`consults`); otherwise, `p` joins the queue `waitps`.

  We record that `p` fell sick in `hist1`. If a consultation `(p,d)` began, we record it in `hist2`.

- A `cure_event`: Some consultation terminates (doctor `d` cures patient `p`). Of course, this can only happen if `consults` is non-empty.

  Effects: We use `r`, a random number in the range 0 to 1 to choose which patient-doctor consultation terminates. The patient `p` rejoins the well patients (`wellps`). If there is a waiting patient `p'` in `waitps`, the doctor `d` goes into consultation with `p'`; otherwise, the doctor rejoins the queue `waitds`.

  We record that `p` was cured in `hist3`. If a consultation `(p',d)` began, we record it in `hist2`.

These state transitions are encoded in the functions `sick_event` and `cure_event` (all the state components are modelled as lists). Each function takes a random number and a state and produces a new state. The sick-patients and free-doctors queues are represented as lists, where the first element of the list represents the head of the queue. Thus, enqueuing is performed using "`++`", the built-in infix list-appending operator. It should be observed that the histories are constructed in reverse order. We shall reverse them at the end of the simulation.

`Simulate` is the top-level driver. It takes a stream of random numbers (in the range 0 to 1), the initial list of patients and the initial list of doctors, and calls `process`, passing in empty lists for `consults`, `waitps`, `hist1`, `hist2` and `hist3`.

The function `process` chooses an action randomly: stop, a sick-event, or a cure-event. In the first case, it reverses the histories and returns them as the final result. In the latter two cases, it applies the appropriate state-transition function, and recursively calls `process` on the new state.

4.2.1 A test run

We ran the following test program:

```
def test (patients,doctors,seed) =
    { randoms = mk_random_stream seed ;
      (ps,pds,cs) = simulate randoms patients doctors ;
    In
      (length ps),(length pds),(length cs),ps,pds,cs } ;
```

supplying a list of 20 patients ("P0" through "P19"), a list of 4 doctors ("D0" through "D3") and an initial random seed. The first three components of `test`'s resulting 6-tuple showed that during the simulation, patients fell sick 92 times, doctors were paired with patients 76 times, and doctors cured patients 72 times.

It is clear from the parallelism profile shown in Figure 7 that the program does not have much parallelism.

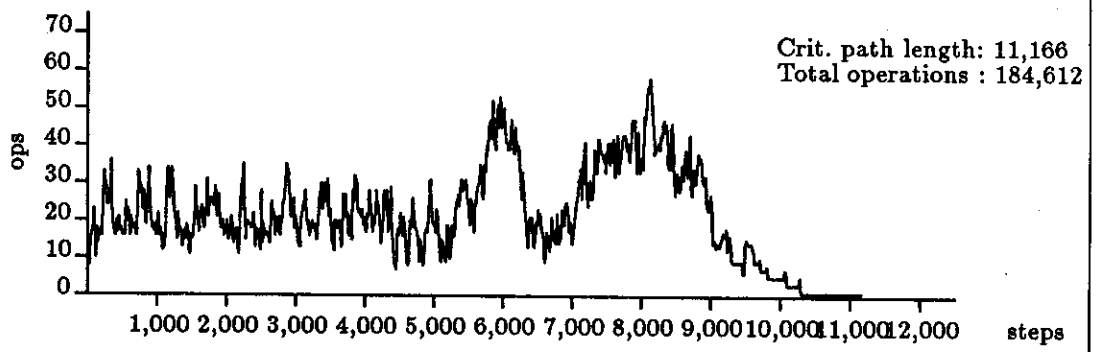

Figure 7: Parallelism profile for determinate Doctor program (20 patients, 4 doctors)

4.3 Non-deterministic programming in Id

As mentioned in the introduction, Id is a layered language. The first two layers (functional, and I-structures) retain determinacy— results depend only on inputs. However, in operating systems code and other applications, we may *require* non-deterministic behavior. The doctor's office problem may be viewed as an abstraction of the resource manager problem in operating systems. The doctors may be regarded as resources that are demanded and held non-deterministically by client processes (patients).[4]

To express such computations, Id has another layer called "managers". Managers are a relatively recent addition to Id. Although the technical ideas behind managers

[4]We take this asymmetric view only because of the wording of the problem statement. If doctors went away for random periods between seeing patients, then we would have symmetry. We would model "consultations" as the resource that is demanded by two kinds of clients (doctors and patients).

are quite well understood, the notation is still experimental and is not yet part of the Id manual. We have a prototype implementation, and a test run of the program described in the next section was executed on this implementation.

There have been various attempts to introduce non-determinism into functional languages. Perhaps the most common approach is to use a special non-deterministic **merge** operation: given two stream inputs, **merge** produces an interleaved output, where the interleaving is non-deterministic. In an actual implementation, the interleaving is typically done in the temporal order in which elements of the input streams become available. In order to distribute results from a resource manager to requestors, it is necessary to: tag the requests in the input-streams with unique stream-identifiers before merging them non-deterministically; carry these tags along with the resource-allocation computation so that they identify which result is meant for which requestor, and split and distribute the output stream according to these tags.

This approach to non-determinism is highly unsatisfactory for several reasons. First, it is very difficult to use when the number of streams to be merged (number of users of a resource) is not manifest, leading to a "spaghetti" of tagging and plumbing. Second, it is very difficult to follow a static type discipline, because all the different kinds of requests to a resource manager, each with different arguments, must be merged into a single homogeneously typed stream.

A more elegant attempt is described in [6], from where we have inherited the term "managers". A manager was specified as a stream-to-stream function, and the tagging and non-deterministic merge at the entry to the manager was hidden with clever notation and clever implementation. While eliminating the "plumbing" problem of explicit non-deterministic merges, the static-type-checking issue still remained.

The Id **manager** construct not only solves all these problems, but also lends itself to very efficient implementation.

Non-determinism and side-effects are closely related— each can be used to simulate the other. However, managers are an attempt to facilitate disciplined use of side-effects in the presence of parallelism. The **manager** construct declares a new, abstract type, *i.e.*, objects of this new type may only be manipulated by the set of interface procedures (called *handlers*) specified in the manager construct. However, unlike ordinary abstract types, these objects have *state* that may be updated by the handlers. Thus, each handler not only specifies a result to be returned, but also possibly a new state for any objects given to it as arguments. Manager semantics guarantee that the state transitions on an object appear to be atomic.

We begin with a simple random-number generator manager that uses `rand_fn`, the linear congruence method from Section 4.1:

```
manager rand_supplier = Cons_cell float
{
    def make_rand_supplier seed = Cons_cell seed ;

    def next_rand (Cons_cell seed) = { r, seed' = rand_fn seed ;
                                       new seed = seed'
                                     In
                                       r } ;
} ;
```

The first line is similar to an algebraic type definition. It defines a new type, `rand_supplier`, and specifies the constructors for this type— here, just one unary constructor `Cons_cell`. By using the keyword `manager` instead of `type`, we indicate (a) that it is updateable and (b) that the constructor `Cons_cell` is visible only in the statements that follow between braces, thus making `rand_supplier` an *abstract type* with respect to the rest of the program. The statements in braces consist of two handlers. The first is a constructor of new `rand_supplier` objects. Given a seed, it creates a new object containing the seed, and returns the new object as its result. This object is first-class, *i.e.,* in the rest of the program, it can be an argument or result of a procedure, it can be stored in data structures, *etc.* However, because `Cons_cell` is not visible outside the manager declaration, the object is *opaque* to the rest of the program. To do anything that requires manipulation of the internal state (reading it or updating it) it has to be passed to one of the handlers, which are the only procedures that can examine and update the internal state.

The second handler, when applied to a `rand_supplier` object, applies `rand_fn` to the old seed value to produce a random number `r` and a new seed `seed'`. The update to the state is specified by the binding that uses the `new` keyword. The procedure returns the value `r`.

Because a `rand_supplier` object is a first-class object, there may be many references to it. Thus, there may be many concurrent attempts to apply `next_rand` to it. Managers guarantee that such concurrent accesses are *serializable*, even though the user has not mentioned any locking or synchronization. The reading-out of the seed, application of `rand_fn` to it, and storing of the new seed is performed as an *atomic* action. Thus, two concurent accessors can never see the same seed.

In general, a manager object can contain multiple components, and each handler may read and update more than one component. Atomicity is still guaranteed, *i.e.,* another concurrent execution of a handler on the same object cannot read intermediate states. These properties of a managed object make it easy to establish and maintain invariants on the state of the object by ensuring that the *each* state update by a handler maintains the invariant.

Thus, managers are akin to Hoare's *monitors* [7]. However, there is an important

difference that has far-reaching consequences. Within a handler, the new value of each state component can be specified at most once. This allows handlers to be *non-strict*, *i.e.*, the state update (critical section) and the return-value computation can be decoupled.

One consequence of this decoupling is increased concurrency. The return-value computation does not have to be in the critical section, so that the critical section may be released before the return-value is computed. Conversely, the critical section does not have to be in the critical path of the caller of a handler, so that a result may be returned to the caller before the critical section is completed.

A second major consequence is that a manager can have complete control over the scheduling of concurrent accessors (events, `wait`'s and `signal`'s, in monitor terminology). To achieve this, we use a feature in Id called "I-structures". I-structures are dynamically allocated and, at first, appear to be ordinary updateable cells. For example, the function `make_cell` allocates a cell, the procedure `put_cell` stores a value in the cell, and the procedure `get_cell` reads a value from the cell. However, I-structure semantics make them different from updateable cells. In particular, a value may be put into a cell *at most once*, and `get_cell` automatically blocks on an empty cell until a value has been written there. An error occurs if an attempt is made to put a value twice into a cell.

We illustrate this device with a manager for a *queue* of strings. A small modification of this example will be used later in our Doctor's office solution. We wish to use handlers `enq` and `deq` to enqueue and dequeue strings from the queue, respectively. A fundamental difference from, say, the `rand_supplier` manager is that the queue manager may have to respond to requests *out of order*. In particular, if one process attempts `deq` on an empty queue, its response must be deferred until another process performs an `enq` operation. We call such managers *scheduling* managers, as opposed to *simple* managers like `rand_supplier`.

The code for the queue manager is shown below.

```
manager queue = Cons_queue (list S)                  % queued strings
                           (list (cell S))           % waiting dequeuers
{
    def make_queue xs = Cons_queue xs Nil ;

    def enq (Cons_queue xs cs) x =                   % x is a string
        {case cs of
           Nil      = { new xs = xs ++ (x:Nil)       % Enter x at end of queue
                      In
                        Ok }
         | (c:cs') = { new cs = cs' ;
                       call put_cell c x             % Send x to the dequeueing
```

```
                    In                                  % process that is blocked on c
                      Ok } } ;

    def deq (Cons_queue xs cs) =
        {case xs of
           Nil      = { c = make_cell _ ;               % Make a cell to block on
                        new cs = cs ++ (c:Nil)          % Enter it at end of dequeuers
                      In
                        get_cell c  }                   % Block, trying to read the cell
         | (x:xs') = { new xs = xs' ;
                      In
                        x } } ;
} ;
```

In the first line, we indicate that the internal representation of a queue object is built with the `Cons_queue` constructor, and contains two components— a list of strings ("`S`" is the type for strings) and a list of cells that can contain strings (cells are typed objects). The first list represents the strings that are currently enqueued. The second list represents the cells on which dequeuing processes are waiting. Note that both lists will never simultaneously be non-empty, *i.e.,* either there can be strings enqueued or there can be waiting processes, but not both.

The constructor `make_queue` takes an initial list of strings `xs`, builds a queue object containing these strings and an empty list of waiting cells, and returns this object.

The `enq` handler checks if there are any waiting processes (cells `cs`). If not, it attaches the given string `x` at the end of the list of enqueued strings (using the built-in list-concatenation operator "`++`"). If there is a waiting process (blocked on cell `c`), it puts the string `x` into the cell (thereby unblocking the waiting process and giving it the string), and updates the list of cells to be the rest of the cells `cs'`. In either case, the constant `Ok` is returned as a result to the enqueueing process. The keyword `call` is used to indicate that the expression following it is executed purely for its side-effect (in this case, the call to `put_cell`).

The `deq` handler checks if there are any enqueued strings `xs`. If not, it allocates an empty cell `c`, appends it to the list of waiting cells, and blocks trying to read the cell `c`. Because of the non-strict evaluation mechanism, the blocking takes place *outside* the critical section, *i.e.,* the state update can take place and the object is then available for other concurrent processes, one of which will presumably unblock this process by writing a value into `c`. Note that we use "`_`" as a "don't care" argument to `make_cell`.

If there is an enqueued string `x` available to a `deq` request, it is returned as a result, and it updates the state to contain the remaining enqueued strings `xs'`.

Thus, the semantics of managers are such that it allows us to choose the order in which it responds to requests. This is in contrast to the *implicit* scheduling imposed

by the `wait` and `signal` constructs in Hoare's monitors.

4.4 A non-deterministic simulator for the doctor's office

From the queue example, it is but a small step to the non-deterministic doctor's office simulator. The code is shown in Figure 8. The top-level function is `simulate`, which takes a random-number supply (an object as discussed above), a list of patients, a list of doctors, and a number `max_iter` which determines the duration of the simulation. It creates a doctor's office initialized with all doctors free (to be described below). The `for` p-loop, being a parallel loop, simulates all patients simultaneously.

The `for` j-loop simulates the life of each patient. Until the end of the simulation[5], patient `p` repeatedly behaves like this. He is healthy for a random duration (using the procedure `delay`) and then falls sick, at which point he asks the office for a doctor. When he gets a doctor, he consults with him for a random duration (second `delay`) and is then cured. He then returns the doctor to the office and repeats the cycle.

The `THEN` separator is used to force a sequencing where otherwise things would have been done in parallel. When all the loops have terminated (ensured by the last `THEN`), we extract and return the histories maintained in the office.

The manager definition for the doctors' `office` is shown next in Figure 8. Like the `queue` manager, it also maintains two queues: a queue of free doctors and a queue of waiting sick patients; in addition, it maintains the three history lists (patients falling sick, patients-and-doctors going into consultation and patients cured). We model a doctor by a string (the doctor's name). A waiting patient is modeled by a string (the patient's name) and a cell that represents the place where a doctor should be put when one becomes available.

The constructor `make_office` simply creates a new `office` object with the free-doctors component initialized to the given list of doctors, and all other components empty.

The handler `get_doctor` checks if there is any free doctor `d`. If so, he is removed from the free list `ds`; `p` and `d` are entered in history `h2`, and `d` is returned to the requestor. If there is no free doctor, then the response is deferred using a cell `c` which is appended with `p` into the list of waiting sick patients. In either case, `p` is entered in history `h1`.

The handler `put_doctor` checks if there is any waiting sick patient `p'`. If so, `p'` is taken off the waiting list, the released doctor is immediately sent to `p'` *via* the associated cell `c`, and `p',d` is entered in `h2`. If there are no waiting patients, the doctor is returned

[5]We have modeled the duration of the simulation as a fixed number of iterations, since the problem statement does not specify anything. Another possibility would be to consult a real-time clock (which would, of course, be implemented as a manager object.)

```
def simulate rand_supply patients doctors max_iter =
    { office = make_office doctors ;
      {for p <- patients do                          % For each patient,
        {for j <- 1 to max_iter do
              delay (next_rand rand_supply) ;        % Be healthy for random time
         THEN d = get_doctor office p ;
         THEN delay (next_rand rand_supply) ;        % Consult for random time
         THEN call put_doctor office p d
         THEN }} ;
    THEN
    In
      histories office } ;

manager office = Make_office (list S)                          % waiting doctors
                             (list (S,cell S))                 % waiting patients
                             (list S) (list (S,S)) (list S)    % histories
{
    def make_office doctors = Make_office doctors Nil Nil Nil Nil Nil ;

    def get_doctor (ds,sps,h1,h2,h3) p =
        {case ds of
           (d:ds') = { new ds = ds' ;                          % Doctor d available
                       new h2 = (p,d):h2 ;  new h1 = p:h1
                     In
                       d }
         | Nil     = { c = make_cell _ ;                       % No doctor available
                       new sps = sps ++ (p,c):Nil ;
                       new h1 = p:h1
                     In
                       get_cell c }} ;

    def put_doctor (ds,sps,h1,h2,h3) p d =
        {case sps of
           ((p',c):sps') = { new sps = sps' ;                  % Patient p' waiting
                             call put_cell c d ;               % Send doctor to him
                             new h2 = (p',d):h2 ;  new h3 = p:h3
                           In
                             Ok }
         | Nil           = { new ds = ds ++ d:Nil ;            % No patient waiting
                             new h3 = p:h3
                           In
                             Ok }} ;

    def histories (ds,sps,h1,h2,h3) = (reverse h1), (reverse h2), (reverse h3) ;
} ;
```

Figure 8: Non-deterministic program for the doctor's office problem.

to the free doctors list. In either case, the cured patient `p` is entered in history `h3`, and an acknowledgment (`Ok`) is returned to him.

Finally, the handler `histories` returns the three histories, first reversing them because they were collected in the reverse order.

4.4.1 A test run

The code we ran was a small variation on the code shown in Figure 8 because our experimental implementation is based on an older notation. The parallelism profile is shown in Figure 9.

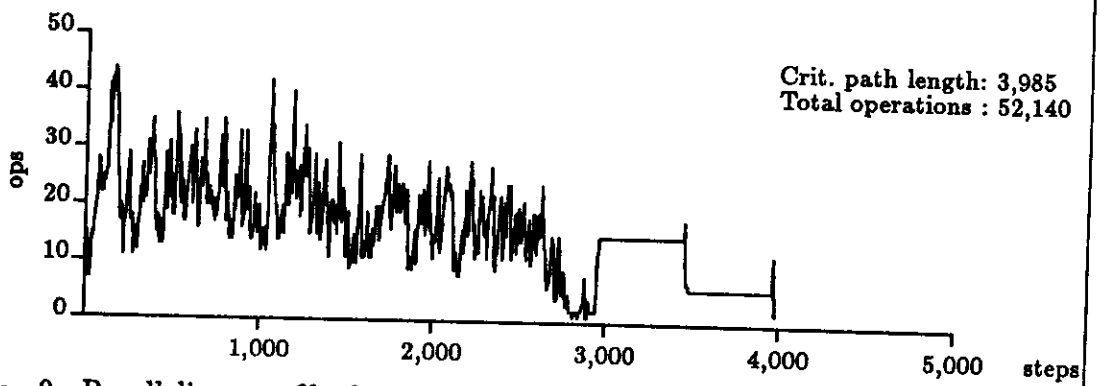

Figure 9: Parallelism profile for non-determinate Doctor program (20 patients, 4 doctors)

5 Skyline matrix solver

5.1 Crout's method for LU decomposition

To solve the system of linear equations $\mathbf{Ax} = \mathbf{b}$, our approach is based on LU decomposition using Crout's method, as described in [8]. We first show a solution for dense matrices, and then modify it for skyline matrices. Suppose we express $\mathbf{A}$ as the product of some $\mathbf{L}$ and $\mathbf{U}$, which are lower and upper triangular matrices, respectively:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

For example ($n = 4$):

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{43} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{32} & a_{33} & a_{34} \\ a_{14} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Then, it is clear that:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{b}$$

So, we can solve for $\mathbf{x}$ in two stages. In the *forward substitution* stage, we find $\mathbf{y}$ such that:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

and then, in the *backward substitution* stage, we find $\mathbf{x}$ such that:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

Each of the last two equations involves triangular matrices, and so the solution is easy. For forward substitution:

$$\begin{aligned} y_1 &= \tfrac{b_1}{l_{11}} \\ y_i &= \tfrac{1}{l_{ii}} \left[b_i - \textstyle\sum_{j=1}^{i-1} l_{ij} y_j \right] \qquad 2 \le i \le n \end{aligned}$$

and, for backward substitution:

$$\begin{aligned} x_n &= \tfrac{y_n}{u_{nn}} \\ x_i &= \tfrac{1}{u_{ii}} \left[y_i - \textstyle\sum_{j=i+1}^{n} u_{ij} x_j \right] \qquad 1 \le i \le (n-1) \end{aligned}$$

Let us consider how to decompose $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$. It is clear from the equation $\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$ that the ij'th element of $\mathbf{A}$ is the inner-product of the i'th row of $\mathbf{L}$ and the j'th column of $\mathbf{U}$, *i.e.*,

$$l_{i1}u_{1j} + l_{i2}u_{2j} + ...l_{in}u_{nj} = a_{ij}$$

However, since l_{ij} is zero whenever $i < j$ and u_{ij} is zero whenever $i > j$, this equation can be separated into two cases (note the final term in each case):

$$\begin{aligned} i \le j: \qquad & l_{i1}u_{1j} + l_{i2}u_{2j} + ...l_{ii}u_{ij} = a_{ij} \\ i > j: \qquad & l_{i1}u_{1j} + l_{i2}u_{2j} + ...l_{ij}u_{jj} = a_{ij} \end{aligned}$$

Further, it is always possible to choose the diagonal elements of $\mathbf{L}$ (*i.e.*, l_{ii}) to be 1. The last two equations can then be rearranged as:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad 1 \le j \le n, 1 \le i \le j$$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \qquad 1 \le j \le n, j+1 \le i \le n$$

Since **L**'s diagonal elements (l_{ii}) are assumed to be one, we do not compute them, and we do not store them. When this diagonal of **L** is omitted, the remaining **L** and **U** elements have disjoint indices. Therefore, they can be stored in a single $n \times n$ matrix called **LU**. This structure, along with the computation of the **L** and **U** elements, is depicted in Figure 10.

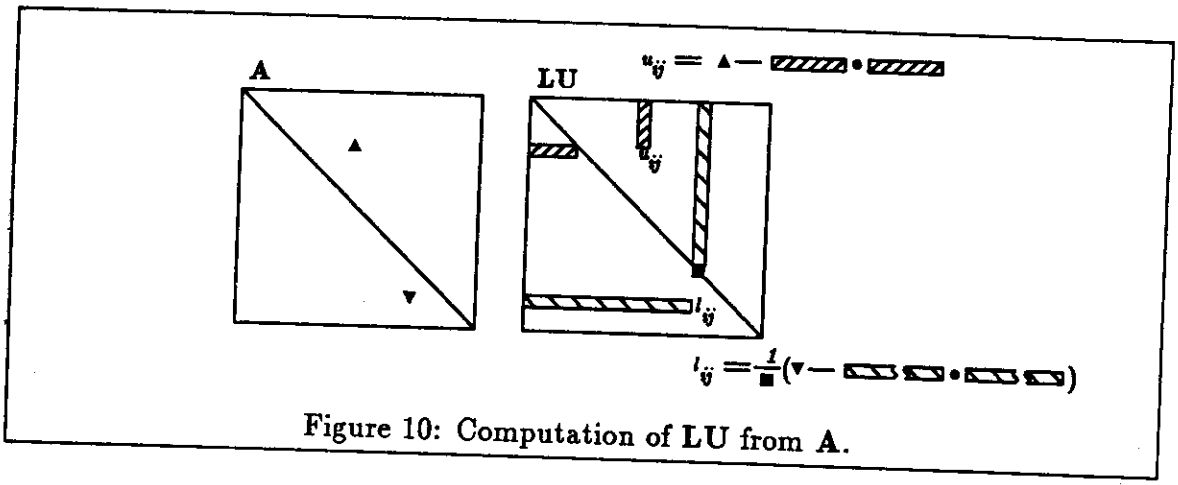

Figure 10: Computation of **LU** from **A**.

The forward and backward substitution computations are depicted in Figures 11 and 12, respectively.

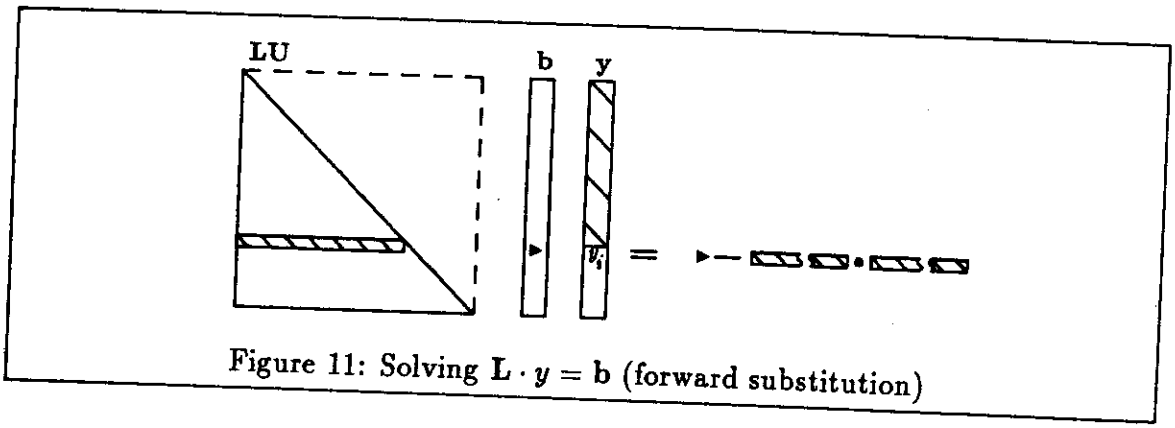

Figure 11: Solving $\mathbf{L} \cdot y = \mathbf{b}$ (forward substitution)

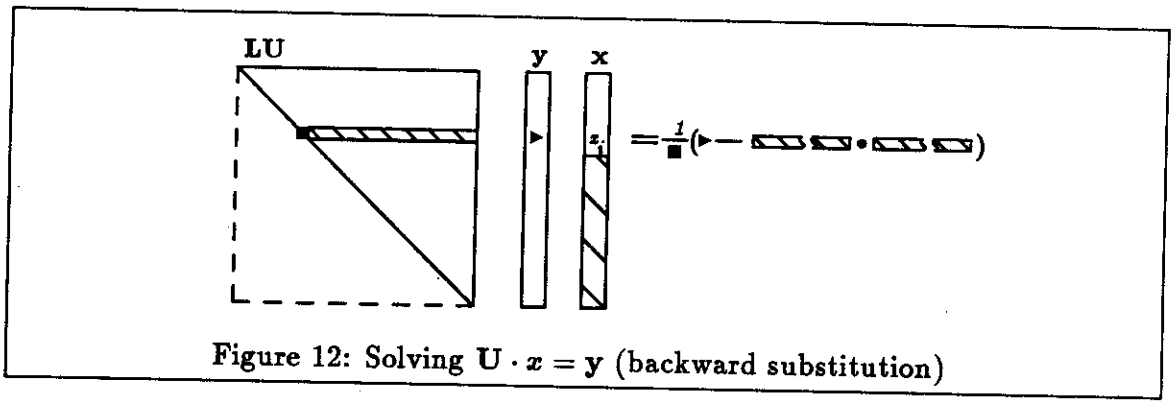

Figure 12: Solving $\mathbf{U} \cdot x = \mathbf{y}$ (backward substitution)

5.2 LU decomposition of dense matrices

The Id code for LU decomposition of a dense matrix follows the equations given above exactly (Figure 10):

```
def LUDCMP_dense A =
  { (_,_),(_,n) = matrix_bounds A ;

    LU = {matrix (1,n),(1,n)
          | [i,j] = ufn i j || j <- 1 to n & i <- 1 to j          % upper
          | [i,j] = lfn i j || j <- 1 to n & i <- (j+1) to n } ; % lower

    def ufn i j = sum_down 1 (i-1) A[i,j] (term i j) ;           % upper

    def lfn i j = (1/LU[j,j]) * (sum_down 1 (j-1) A[i,j] (term i j)) ; % lower

    def term i j k = LU[i,k]*LU[k,j] ;
   In
     LU } ;
```

In the first line in the block, we find n, the dimension of the problem. The primitive function `matrix_bounds` returns the index bounds of **A**, represented as a pair of pairs. The pattern on the left-side of the binding shows this pair-of-pairs structure, and ignores three of the components (using "_" for a "dont-care" pattern) and binds `n` to the fourth. Here, we are assuming that `A` has bounds (1,n),(1,n)— it would be quite easy to bind all four components and verify that this is indeed so.

The second binding defines the actual **LU** matrix, using Id's array comprehension notation. It specifies the contents of the 2-dimensional array in two regions corresponding to **U** and **L**, respectively, using the functions `ufn` and `lfn`. Each expression

of the form (sum_down k_1 k_2 a f) computes

$$a - \sum_{k=k_1}^{k_2} f(k)$$

the Id code for which is:

```
def sum_down k1 k2 a f = {for k <- k1 to k2 do
                            next a = a - (f k)
                          finally a } ;
```

Here is the code for the forward and backward substitutions. Again, the code is self-evident, corresponding exactly to the equations and Figures 11 and 12.

```
def LUBKSB_dense LU B =
  { (_,n) = bounds B ;

    Y = {vector (1,n)
         | [1] = B[1]
         | [i] = sum_down 1 (i-1) B[i] (yfn i) || i <- 2 to n} ;

    def yfn i j = LU[i,j] * Y[j] ;

    X = {vector (1,n)
         | [n] = Y[n] / LU[n,n]
         | [i] = (1/LU[i,i])*
                 (sum_down (i+1) n Y[i] (xfn i)) || i <- (n-1) downto 1 } ;

    def xfn i j = LU[i,j] * X[j] ;
  In
    X } ;
```

5.3 LU decomposition of skyline matrices

In Figure 10, in each inner-product, note that the low index of each component vector is always 1. In a skyline matrix, on the other hand, the low index of the horizontal vector will be some $i1 \geq 1$ and the low index of the vertical vector will be some $j1 \geq 1$. Thus, the inner product can be "clipped" to begin at $max(i1, j1)$.

A second important observation is that it is clear from the equations that the **LU** matrix will always have exactly the same skyline shape as the original **A** matrix, so that the data structure for **LU** can be identical to the data structure for **A**.

The data structures that we choose for **A** (and **LU**) is shown in Figure 13. The sub-diagonal elements of **A** are held in **AL**, which is an n-vector of vectors. The i'th row is represented by a vector with dimensions $(j1, i-1)$, where $1 \leq j1 \leq i$ is the minimum

index. When $j1 = i$, the lower index is greater than the lower index, representing an *empty* row vector. These are depicted by little circles "o" in the figure. The diagonal and super-diagonal elements of **A** are held in **AU**, which is also an n-vector of vectors. The j'th row is represented by a vector with dimensions $(j1, j)$, where $1 \leq j1 \leq j$ is the minimum index. Note that none of the column vectors can be empty.

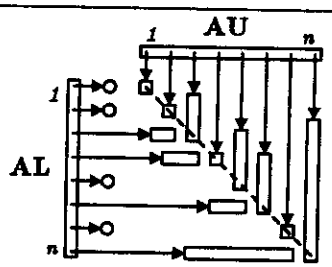

Figure 13: Data structure for **A** (**LU** is similar).

The code for the decomposition function is shown in Figure 14. In the definition of `U`, the j'th column vector is specified as a vector with the same bounds as the j'th column vector in `AU` (from `i1` to `j`). In calling `ufn`, we supply it `i1` and `j1`, the lowest indices of the column vector and row vector of the inner-product. In `ufn`, we clip the iteration to begin at max(`i1`, `j1`). A similar strategy is used in the specification of `L`.

The code for forward and backward substitution is shown in Figure 15. Recall that in the dense version, `yfn` was defined thus:

```
def yfn i j = LU[i,j] * Y[j] ;
```

Now, however, the term `LU[i,j]` must be replaced by `L[i][j]`. However, `L[i]` is fixed for each `sum_down` traversal, and so we optimize it by passing the entire row vector `L[i]` to `yfn`, which then just indexes it with `j`.

In `xfn`, we run across the following problem: the inner-product traverses a *row* of `U`. However, because of our representation of the skyline `U`, not all elements in a particular row may be present. So, our code first extracts the j'th column vector, and then extracts the index bounds of that vector. If i is outside the bounds, the term is 0; otherwise, we extract the normal $LU[i,j]$ (*i.e.*, $U[j][i]$, which is `Uj[i]`).

Of course, this conditional is executed $O(n^2)$ times (once for every position in the upper triangle). We could trade time for space by first reformatting `U` into a full upper triangular matrix (filling in all the zeroes) and then indexing it as usual. We do not pursue this possibility here.

Finally, the top-level function to solve a given set of equations is shown in Figure 16, where we assume that `A` is a pair of skylines `(AL,AU)`.

```
def LUDCMP_sky (AL, AU) =
    { (_,n) = bounds AU ;

      U = {vector (1,n)
           | [j] = { (i1,_) = bounds  AU[j]
                   In
                     {vector (i1,j)
                      | [i] = { (j1,_) = bounds  AL[i]
                              In
                                ufn i1 j1 i j}       || i <- i1 to j}}
                                                        || j <- 1 to n} ;

      L = {vector (2,n)
           | [i] = { (j1,_) = bounds  AL[i]
                   In
                     {vector (j1,i-1)
                      | [j] = { (i1,_) = bounds  AU[j]
                              In
                                lfn i1 j1 i j}       || j <- j1 to (i-1)}}
                                                        || i <- 2 to n} ;

      def ufn i1 j1 i j = sum_down  (max i1 j1)  (i-1)  AU[j][i]  (term i j) ;

      def lfn i1 j1 i j = (1/U[j][j]) *
                          (sum_down  (max i1 j1)  (j-1)  AL[i][j]  (term i j)) ;

      def term i j k = L[i][k] * U[j][k] ;
    In
      L,U } ;
```

Figure 14: LU decomposition for skyline matrices.

```
def LUBKSB_sky (L,U) B =
  { (_,n) = bounds B ;

    Y = {vector (1,n)
         | [1] = B[1]
         | [i] = { (j1,_) = bounds  L[i]
                   In
                     sum_down j1 (i-1) B[i] (yfn L[i] i)} || i <- 2 to n} ;

    def yfn Li i j = Li[j] * Y[j] ;

    X = {vector (1,n)
         | [n] = Y[n]  / U[n][n]
         | [i] = (1/U[i][i])*
                 (sum_down (i+1) n Y[i] (xfn i)) || i <- (n-1) downto 1 } ;

    def xfn i j = { uj = U[j] ;
                    (i1,_) = bounds uj ;
                    In
                      if (i < i1) then
                        0.0
                      else
                        Uj[i] * X[j] } ;
  In
    X } ;
```

Figure 15: Forward and backward substitution for skyline matrices.

```
def solve_sky (A,B) = LUBKSB_sky (LUDCMP_sky A) B ;
```

Figure 16: Top-level skyline matrix solver.

5.4 Test runs

We generated a random 50 × 50 `A` matrix (*i.e.*, picked a random envelope and filled with random numbers) containing 1210 elements, *i.e.*, a density of about 50%. We also generated a randomly-filled `B` vector of size 50. Figure 17 shows the parallelism profile generated by GITA when `solve_sky` is run on these inputs. Figures 18 and 19 show the individual contributions of `ludcmp_sky` and `lubksb_sky`, respectively, to the composite parallelism profile.

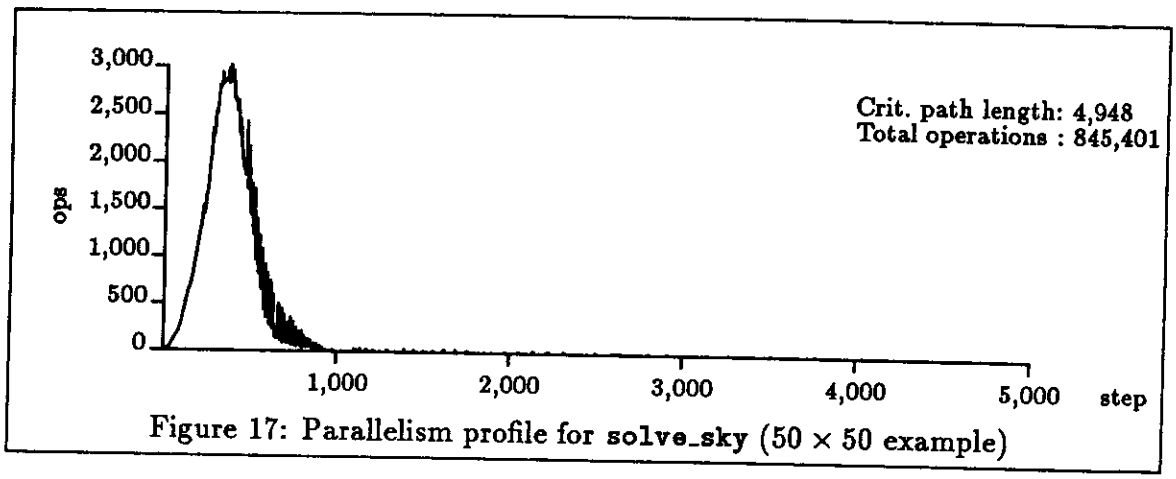

Figure 17: Parallelism profile for `solve_sky` (50 × 50 example)

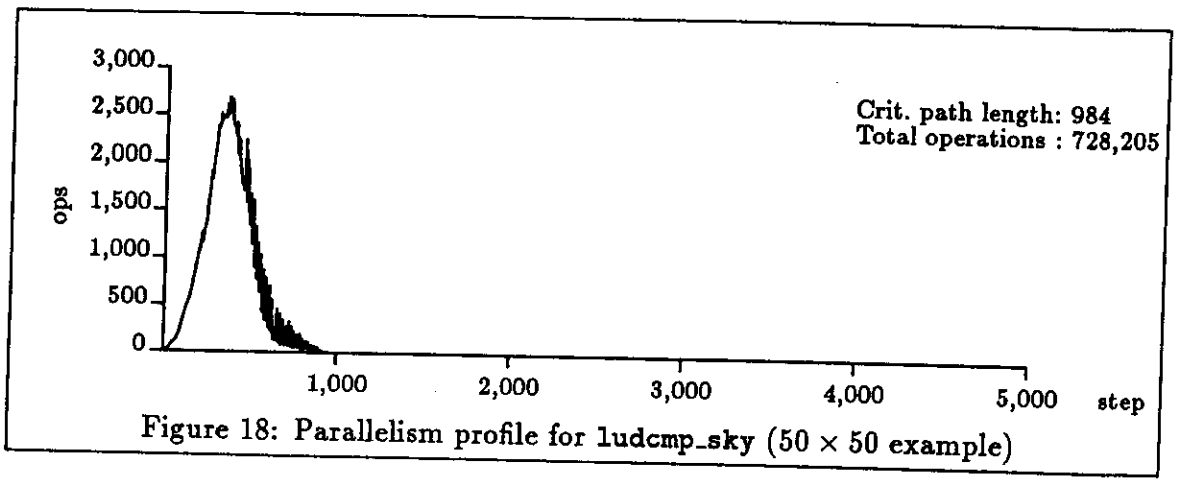

Figure 18: Parallelism profile for `ludcmp_sky` (50 × 50 example)

It is clear that almost all the parallelism in the skyline matrix solver is in the LU decomposition stage. This is actually quite obvious if we analyze Crout's algorithm, which exhibits "wavefront" parallelism. Each element in the LU matrix depends only on those above it and to the left, so that a frontier that is perpendicular to the diagonal can be computed in parallel. This frontier sweeps across the matrix like a wave from the top left to the bottom right.

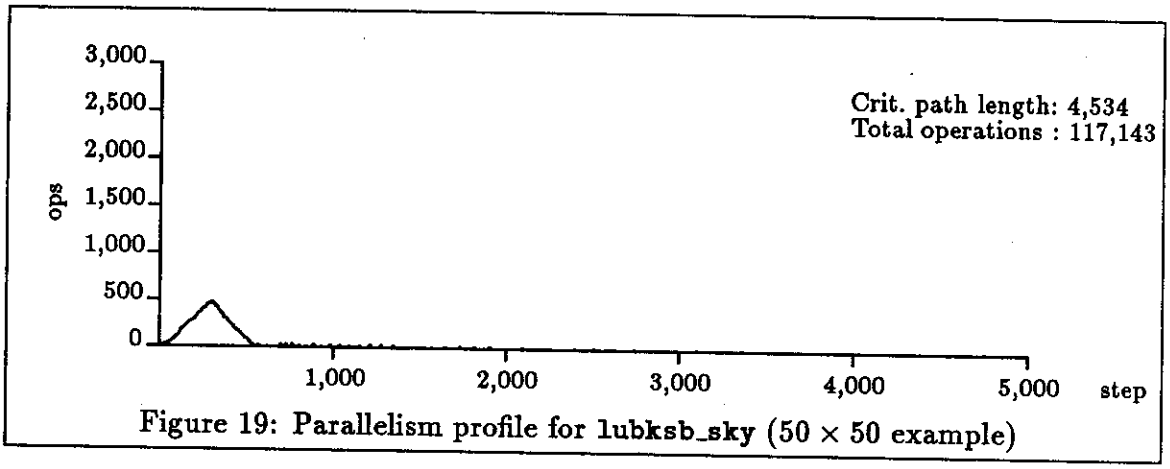

Figure 19: Parallelism profile for `lubksb_sky` (50×50 example)

In the forward and backward substitution stages, since the `y` and `x` matrices are filled using linear recurrences, there is hardly any parallelism at all.

In languages with explicit parallelism, the wavefront parallelism of LU decomposition could be expressed with a little effort for dense matrices. We could have a sequential loop that iterates down the diagonal, and a nested, parallel loop that computes all the cross-diagonal elements in parallel. Unfortunately, it does not appear so easy to extend such a solution to skyline matrices, because of the irregular shape of the cross-diagonals. In Id, on the other hand, implicit parallelism gives us automatic producer-consumer synchronization, allowing it to adapt dynamically to such irregular parallel structures.

Acknowledgements

Steve Heller was mainly responsible for our solution to the paraffins problem, developed in 1988.

The non-deterministic doctor's office program was developed with much discussion with Paul Barth, who is also responsible for the implementation of managers and performing our test run.

Although our skyline matrix solver uses Crout's method, we have learned a lot from previous work on linear equation solvers that used Gauss elimination. K. Ekanadham of IBM Research developed elegant Id programs for dense matrices, and Javed Aslam, Christopher Colby and Ken Steele developed Id programs for sparse matrices.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

References

[1] R.S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[2] R.S. Nikhil and Arvind. *Programming in Id: a parallel programming language.* 1990. (book, in preparation).

[3] Arvind, S.K. Heller, and R.S. Nikhil. *Programming Generality and Parallel Computers*, pages 255–286. ESCOM Science Publishers, P.O.Box 214, 2300 AE Leiden, The Netherlands, September 1988. Proceedings of the Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy.

[4] D.E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley, Reading, Massachusetts, 1973.

[5] D.E. Knuth. *The Art of Computer Programming, Volume 2: Semi-Numerical Algorithms.* Addison Wesley, Reading, Massachusetts, 1973.

[6] Arvind and J. Dean Brock. Resource Managers in Functional Programming. *Journal of Parallel and Distributed Computing*, 1(1), June 1984.

[7] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[8] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[9] D.A. Turner. The semantic elegance of applicative languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92, October 1981.

A Problem statements

The following are the statements of the four problems addressed in this paper.

A.1 Hamming's problem, extended

Given a set of primes $\{a, b, c, \ldots\}$, of unknown length, and an integer n, output in increasing order and without duplicates all integers of the form

$$a^i \cdot b^j \cdot c^k \times \ldots \qquad \leq n$$

A.2 The paraffins problem

The chemical formula for paraffin molecules is C_iH_{2i+2}. Given an integer n, output without repetition and in order of increasing size the structural representation of all paraffin molecules for $i \leq n$. Include all isomers, but no duplicates. You may choose any representation of the molecules you wish, so long as it clearly distinguishes among isomers. The problem is discussed by Turner in

> Turner, D.A., "The Semantic Elegance of Applicative Languages", *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, October 1981, *pp.* 85-92.

Note Turner's solution is inefficient. To generate paraffins without duplicates, see the discussion of enumerating trees in:

> Knuth, D.E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley, Reading, Massachusetts, 1973.

A.3 A doctor's office

Given a set of patients, a set of doctors, and a receptionist, model the following interactions: initially, all patients are well, and all doctors are in a queue awaiting sick patients. At random times, patients become sick and enter a queue for treatment by one of the doctors. The receptionist handles the two queues, assigning patients to doctors in a first-in, first-out manner. Once a doctor and patient are paired, the doctor diagnoses the illness and, in a randomly chosen period of time, cures the patient. The patient is then released until such time as he or she becomes sick again, and the doctor rejoins the queue to await another patient. Return a list of patients in the order they become sick, a list of patient/doctor tuples in the order patients are assigned doctors, and list of patients in the order they are cured.

This is not a time-driven simulation. There is no a priori knowledge of when events are to occur; there is no global clock, no global communication, and no global knowledge. You may use any distribution functions you wish to decide when a patient becomes sick and how long a patient sees a doctor. The interactions of the patients, doctors and receptionist should be true to life. The purpose of this problem is to see how each language expresses a set of concurrent processes that interact asynchronously and must respond to asynchronous events.

A.4 Skyline matrix solver

Solve the system of linear equations,

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

without pivoting, where $\mathbf{A}$ is an $n \times n$ skyline matrix. A skyline matrix has nonzero elements in column j in rows $1 \leq i \leq j$, and has nonzero elements in row i in columns $1 \leq j \leq i$. The first constraint defines the skyline above the diagonal, which is towards the top, and the second constraint defines the skyline below the diagonal, which is towards the left. The shape of a skyline matrix is depicted in Figure 1 (only the shaded region contains non-zeroes).

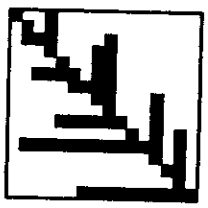

Figure 1: The shape of a skyline matrix.