

# Iterative and Parallel Algorithm Design from High Level Language Traces

Daniel E. Cooke and J. Nelson Rushton

Computer Science Department, Texas Tech University, Lubbock, Texas, U.S.A. 79409  
{dcooke, nrushton}@coe.ttu.edu

**Abstract.** We present a high level language called SequenceL. The language is declarative, in the sense that the programmer describes functions in terms of abstract relationships between their inputs and outputs, rather than in terms of algorithms for carrying out computations. The semantics of the language are capable of automatically discovering and implementing the required algorithms, including iterative and parallel control structures in many cases. Current implementations are prototypes only, and do not produce code of comparable efficiency to that of a good human programmer. Current implementations can, however, be used as a tool to guide human programmers in discovering and comparing options for parallelizing their solutions. This paper describes the language and approach, and demonstrates the use of SequenceL to discover options for parallelizing matrix computations.

## 1 Introduction and Related Work

SequenceL is a compact, Turing-complete, high-level language, in which algorithms for implementing a solution are automatically derived from a high level description of that solution. In this paper we introduce how the language can be exploited in the design of iterative and parallel algorithms. Using SequenceL code paired with sample data, one can run and trace the algorithms derived by our interpreter. These language traces can be used to guide less experienced programmers when they are developing high performance, parallel programs, reducing the difficulty of their task. We demonstrate the approach using two examples. First, we show a simple matrix multiply, followed by the solution of a Partial Differential Equation using a Jacobi Iteration method. The end of the paper indicates how recursion is accomplished.

SequenceL is a small language with a simple semantic, which discovers and evaluates many iterative and all parallel algorithms implied by a high level specification. (See the appendix for the full SequenceL grammar.) The beginning point of this effort can be traced to [3], which overlaps the iterative morphisms found in [9]. SequenceL's semantic work-horse is the *Normalize-Transpose-Distribute* (NTD), which is used to simplify and decompose structures, based upon a dataflow like execution strategy similar to GAMMA [2] and NESL [1]. The NTD semantic achieves a goal similar to that of the Lämmel and Peyton-Jones, boilerplate elimination. [7,8]

## 2 Normalize-Transpose-Distribute (NTD)

SequenceL functions have typed arguments, where the type of an argument is simply its level of nesting (scalar, sequence of scalars (i.e. vector), sequence of sequences of scalars, (i.e., matrix) etc.). When a SequenceL function receives one or more arguments, which are *overtyped* (i.e., nesting level higher than expected), the remaining arguments are *normalized* -- duplicated to match the length of the overtyped arguments. For example, if the + operation, which expects scalars for its first and second arguments, is invoked in the expression  $(1,2,3) + 10$ , normalization results in  $(1, 2, 3) + (10, 10, 10)$ . The normalized arguments are then gathered into a list *transposed*, in this case yielding  $((1,10), (2,10), (3,10))$ . Finally the operation, in this case +, is *distributed*, or performed on the members of the resulting sequence, here yielding  $(11,12,13)$ . The initial normalization has no effect if all of the arguments are overtyped and of the same length.

For another example, consider the definition of dot product in SequenceL:

```
dotprod: vector * vector -> scalar
dotprod(A,B) = sum(A*B)
```

Given the call  $dotprod((2,3,4) (100,10,1))$ , instantiating the function body results in  $sum(((2,3,4) * (100,10,1)))$ . Since \* expects scalars for both arguments, NTD is performed, resulting in  $sum(200, 30, 4)$ , which evaluates to 234, the desired result (sum is a built in function which adds the members of a vector).

Our experience has indicated that a great number, and perhaps the majority, of parallelizable for-loops can be omitted in SequenceL due to this simple semantic transformation. This has the advantage of preventing the errors that often accompany loops, such as initialization and off-by-one errors. It also makes the code more compact, eliminating distractions of boilerplate loops and allowing the programmer to better focus on the “meat” of the computation, both in writing and debugging. Finally, while the solution is more compact and less strenuous to obtain even than a standard sequential implementation, our interpreter can automatically discover and implement parallelisms inherent in the algorithm.

## 3 The Environment and the Matrix Multiply

Although SequenceL was introduced in 1996, [4,5,6] substantial simplifications to the language’s syntax and semantics have been made in the past two years. The size constraints for this paper limit the extent to which we are able to fully introduce the vastly improved version of the language. However, the examples shown here exercise the full capabilities of the language, and several important aspects of the language can be understood from these examples.

From the standpoint of specification, matrix multiplication involves taking the dot product of every *i*’th row of a matrix *a* with each *j*’th column of a matrix *b*, resulting

in the  $i,j$ 'th element of the product matrix. In terms of combining data structures one can view the specification as combining a single row of  $a$  with the transpose of  $b$ :

```
mmrow: vector * matrix → matrix
mmrow(A,B) ::= dotprod(A,transpose(B))
```

This function is written to compute a single row of the matrix product  $AB$ , taking the dot product of the vector  $A$  with each column of the matrix  $B$ . NTD eliminates the need for a for-loop here – dot product expects a vector in its second argument but instead receives a matrix  $transpose(B)$ , and so the operation is automatically performed on respective members of  $transpose(B)$ , which are the columns of  $B$  as specified above. NTD also gathers the resulting scalars into a vector, which is the desired row of the product matrix.

Once the code is written, as above, to compute a single row of the matrix product, the job in SequenceL is finished. This is again due to the NTD semantic. To multiply matrices  $X$  and  $Y$ , we call  $mmrow(X,Y)$ , which expects a vector in its first argument but receives a matrix. An ensuing NTD applies  $mmrow$  to  $Y$  separately with each row of  $X$ , producing the respective rows of the resulting data structure, which is precisely the matrix multiply operation. Here is the trace of a sample execution:

```
(mmrow, (((1, 2, 4), (10, 20, 40), (11, 12, 14)), ((1, 2, 4), (10, 20, 40), (11, 12, 14)))) (1)
```

Since the first argument is a matrix it is decomposed to its three constituent rows. Normalize then makes three copies of the function name and three copies of the second matrix:

```
(mmrow, mmrow, mmrow (2)
((1,2,4),(10,20,40),(11,12,14)),
((1,2,4),(10,20,40),(11,12,14)),((1,2,4),(10,20,40),(11,12,14)),((1,2,4),(10,20,40),(11,12,14)))
```

Next a transpose is performed and three tasks that can be solved in parallel or iteratively:

```
( (mmrow,((1,2,4),(1,2,4),(10,20,40),(11,12,14))), (3)
(mmrow,((10,20,40),(1,2,4),(10,20,40),(11,12,14))),
(mmrow,((11,12,14),(1,2,4),(10,20,40),(11,12,14)))) )
```

Now the language interpreter instantiates the body of the  $mmrow$  function and the transposes are performed (we write  $dp$  for  $dotprod$  to save space):

```
( (dp,((1,2,4),(transpose,((1,2,4),(10,20,40),(11,12,14))))), (4)
(dp,((10,20,40),(transpose,((1,2,4),(10,20,40),(11,12,14))))),
(dp,((11,12,14),(transpose,((1,2,4),(10,20,40),(11,12,14)))))) )
```

After the transposes, the SequenceL  $dp$  function as defined in section 2 is invoked:

```
( (dp,((1,2,4),(1,10,11),(2,20,12),(4,40,14))), (5)
(dp,((10,20,40),(1,10,11),(2,20,12),(4,40,14))),
(dp,((11,12,14),(1,10,11),(2,20,12),(4,40,14)))) )
```

Note that  $dp$  takes two vectors as input. From where we left off in step 5 of the trace, we know that the second argument of each  $dp$  reference is a two-dimensional structure, so another  $NTD$  is performed on each  $dp$  resulting in 9  $dp$  references:

$$\begin{aligned} &(((dp,((1,2,4),(1,10,11))), & (dp,((1,2,4),(2,20,12))), & (dp,((1,2,4),(4,40,14))), & (6) \\ &((dp,((10,20,40),(1,10,11))), & (dp,((10,20,40),(2,20,12))), & (dp,((10,20,40),(4,40,14))), \\ &((dp,((11,12,14),(1,10,11))), & (dp,((11,12,14),(2,20,12))), & (dp,((11,12,14),(4,40,14)))) \end{aligned}$$

The function bodies are now instantiated. From this point, the trace of each  $dp$  is identical to the evaluation of  $dp$  as demonstrated in section 2. E.G., Taking the first  $dp$  from step 6 we have an  $NTD$  followed by the specified arithmetic:  $sum((1,2,4) * (1,10,11)) = sum(1*1, 2*10, 4*11) = sum(1,20,44)$ , resulting in 65, which will take the row 1, column 1 place of the final result:

$$(((65, 90, 140), (650, 900, 1400), (285, 430, 720)))$$

The parallelisms discovered between step 6 and the final result are micro-parallelisms, and therefore, should be carried out iteratively. As it turns out, steps 5 and 6 of the trace are of keen interest in designing parallel or concurrent algorithms. Either point could be used as the basis for the algorithms. In the next two subsections we evaluate these options and how they can be realized in concurrent JAVA codes.

### 3.1 Parallelizing based on step 6 of the trace

Step 6 indicates that we are combining every element of each row of the first matrix with every element of each column of the second, forming a cartesian product of sorts from the two matrices. To accomplish this in concurrent JAVA we would have the following code segments, extracted from a total of about 60 lines of code.

```
... public void run()      {
    s = 0;
    for (k=0;k<=m.length-1;k++)
        { s += m[rs][k] * m[k][cs]; } }
public static void main (String args()) { ...
    for(i=0;i<=(r*c)-1;i++)
        {mat[i].start();          ... } }
```

Notice that the  $\sim O(n^2)$  algorithm that forks the  $run$  threads is in the main program, and that we are indeed computing each element of the result concurrently. This loop follows a nested  $\sim O(n^2)$  algorithm that initialized each of the threads with the proper subscript values for  $rs$  and  $cs$ .

### 3.2 Parallelizing based on step 5 of the trace

Step 5 of the SequenceL trace indicates another option for the parallelization of the desired computation. In this approach, each row of the first matrix is combined with all columns of the second. This suggests moving the  $\sim O(n^2)$  algorithm into the concurrently executed *run* threads, reducing the overhead involved in separating threads and spawning concurrent tasks from  $\sim O(n^2)$  to  $O(n)$ .

```
... public void run()      {
    s = new int(m.length);
    for (rs=0;rs<=m.length-1;rs++)
        for (k=0;k<=m.length-1;k++)
            {s[rs]+=m[rs][k]*m[k][cs];}
public static void main (String args()) { ...
    for(i=0;i<=r-1;i++)
        {mat[i].start();}          ... }
```

This approach concurrently computes the result of each row of the resultant matrix and follows a rule of thumb in developing concurrent codes: parallelize outer loops.

It is interesting to note that though this implementation is substantially more efficient than the previous one, it did not occur to us until after we examined the trace of matrix multiplication in SequenceL. We are, of course, not experts in hand crafting parallel codes, but the point is we gained a little more expertise through experiments with the SequenceL interpreter. Moreover, in terms of finding parallelisms, we conjecture that even for experts it might sometimes be less strenuous to specify a computation in SequenceL (which for matrix multiply, for example, was four lines of code including function signatures) and examine its trace, than to search for possible parallel algorithms by hand.

## 4 Jacobi Iteration

In the next example, we design a concurrent version of a solution to a Partial Differential Equation, utilizing a Jacobi Iteration approach. We focus on the transition step, in which the successive approximation  $\mu'$  is derived from the previous one  $\mu$ :

$$\mu_{j,k}' = \frac{1}{4} (\mu_{j+1,k} + \mu_{j-1,k} + \mu_{j,k+1} + \mu_{j,k-1}) - (\rho_{j,k} \cdot (\frac{1}{4} \Delta^2))$$

This approach leaves the borders of the matrix as they are, and computes each interior element from its neighbors above and below, and on each side. In terms of combining data structures, we notice that we will again take a cartesian product of a matrix and its transpose and can utilize an approach similar to the matrix multiply. However, two major differences arise in this problem. The first is that we will apply the computation selectively (i.e., leaving the borders as they are), as opposed to the more uniform application of the matrix multiply. The second difference is the additional step of an

element-wise subtraction of the modified  $\rho$  from the modified  $\mu$ . The resulting SequenceL code appears as follows.

```
jacobi: matrix * scalar * matrix → matrix
jacobi(a, delta, rho) ::=
  neighbors(a, transpose(a), (1,...,size(a)))-(rho*delta^2)/4

neighbors: vector * matrix * scalar → matrix
neighbors(a,b,i) ::= hlp(a,b,i,(1,...,size(b)))

hlp: vector * vector * scalar * scalar → vector
hlp(a, b, i, j) ::=
  a(j) when (i=1 or size(a)=i) OR (j=1 or size(a)=j)
  else
  (a( i+1) + a( i-1) + b( j+1) + b( j-1)) / 4
```

Each of  $\mu$ 's rows is combined with each of its columns, with *neighbors* playing the role of the *mmrows* function and *hlp* playing the role of *dp*. Besides forming the desired cartesian product of the matrices, the *NTD* also captures the row indices in *neighbors*, and the column indices in *hlp*, so that the  $\mu$  computations can be selectively applied. The trace reveals that once the subscripts are captured and the rows have been combined with the columns, the selective computations can be performed in *hlp*, while the  $\rho$  computations are independently performed – see the last line of the trace below.

```
(hlp((1, 2, 4), (1, 10, 11), 1, 1), hlp((1, 2, 4), (2, 20, 12), 1, 2), hlp((1, 2, 4), (4, 40, 14), 1, 3)),
(hlp((10, 20, 40), (1, 10, 11), 2, 1), hlp((10, 20, 40), (2, 20, 12), 2, 2), hlp((10, 20, 40), (4, 40, 14), 2, 3)),
(hlp((11, 12, 14), (1, 10, 11), 3, 1), hlp((11, 12, 14), (2, 20, 12), 3, 2), hlp((11, 12, 14), (4, 40, 14), 3, 3)) -
(((0.000169, 0.000338, 0.000676), (0.00169, 0.00338, 0.00676), (0.001859, 0.002028, 0.002366))/4)
```

The fact that the  $\rho$  computations are performed independently suggests that we can improve the performance of the JAVA code by tucking the  $\rho$  computations inside the  $\sim O(n^2)$  *run* method, which is also computing each row of the  $\mu$  computations concurrently. Space constraints prevent us from showing other options for parallelizing the JAVA code revealed in the SequenceL trace. Suffice it to say the following code segments, from a total of around 80 lines of code, capture a very good approach to the JAVA solution.

```
... public void run() {
s = new double (m.length);
for (cs=0;cs<=m.length-1;cs++)
  {if((rs!=0)&&(rs!=m.length-1)&&(cs!=0)
      &&(cs!=m.length-1))
    {s[cs]=((m[rs][cs+1]+m[rs][cs-1]+m[rs+1][cs]+
             m[rs-1][cs])/4)-((p[rs][cs]*(d*d))/4);}
  else
    {s[cs]= m[rs][cs]-((p[rs][cs]*(d*d))/4); }}}
```

```
public static void main (String args()) { ...
for(i=0;i<=r-1;i++)
    {mat[i].start();}          ... }
```

## 5 Recursion in SequenceL

Recursion in SequenceL differs from most languages in terms of appearance and evaluation. A SequenceL function unconditionally (in the absence of a *when* clause) or conditionally (in the presence of *when* clause(s)), specifies what is to be evaluated once the function completes execution. In doing so, a function can make reference to any SequenceL operator or user-defined function symbol, including a reference to itself. Because there is no true calling or returning in a function, conventional activation records are unnecessary, reducing the runtime overhead. This is a byproduct of the SequenceL abstraction and is identical to the goal of continuations in functional programming. [10] Consider a recursive definition of *factorial* in SequenceL:

```
fact: s → s
fact(n) ::= fact(n-1) *n when n>0 else 1
```

Here is a trace of the evaluation of  $3!$

```
(fact(3-1) * 3 when 3 > 0 else 1) =
( fact(2) * 3 ) =
( (fact(2-1) * 2 when 2 > 0 else 1) * 3) =
( fact(1) * 2 * 3 ) =
( (fact(1-1) * 1 when 1 > 0 else 1) * 2 * 3) =
( fact(0) * 1 * 2 * 3 ) =
( (fact(0-1) * 0 when 0 > 0 else 1) * 2 * 3) =
( 1 * 1 * 2 * 3 ) = (6)
```

Notice that the build up of multipliers 3, 2, and 1 to the right (in the highlighted steps) in the tableau is effectively like the building of continuations.

## 6 Discussion

Since completing the changes to the language last year, we have conducted additional experiments to discover parallel algorithms for problems including Gaussian Elimination, Quicksort, Discrete Wavelet Transforms, Newton-Raphson solutions to linear equations, among others. We have also investigated applications to remote sensing problems and to the prototyping of Guidance, Navigation, and Control problems for processing onboard the Space Shuttle. These experiments have shown SequenceL to be an easy-to-use and promising tool for exploring ideas and problem solutions in a variety of domains.

## References

1. Guy Blelloch, "Programming Parallel Algorithms," March, 1996, Vol. 39, No. 3. *Communications of the ACM*, pp. 98-111.
2. J-P Banater and D. Le Metayer, "Programming by Multiset Transformation, January 1993, Vol. 36 No. 1. *Communications of the ACM*, pp. 98-111.
3. Daniel E. Cooke and A. Gates, "On the Development of a Method to Synthesize Programs from Requirement Specifications," *International Journal on Software Engineering and Knowledge Engineering*, Vol. 1 No. 1 (March, 1991), pp. 21-38.
4. Daniel E. Cooke, "An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience*, Vol. 26 (11) (November, 1996), pp. 1205-1246.
5. Daniel E. Cooke and Per Andersen, "Automatic Parallel Control Structures in SequenceL," *Software Practice and Experience*, Volume 30, Issue 14 (November 2000), pp. 1541-1570.
6. D. E. Cooke, J. N. Rushton, et. al. "Normalize, Transpose, and Distribute: A Basis for Decomposition and Parallel Evaluation of Nonscalars," in revision *ACM Transactions on Programming Languages and Systems*.
7. R. Lämmel and S. Peyton-Jones, "Scrap your boilerplate: a practical design pattern for generic programming," in Proceedings of TLDI 2003, ACM Press.
8. R. Lämmel and S. Peyton-Jones, "Scrap more boilerplate: reflection, zips, and generalised casts," to appear in Proceedings of ICFP 2004, ACM Press.
9. Meijer, E. and Fokkinga, M.M. and Paterson, R., "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" *FPCA* (Springer-Verlag, 1991) LNCS Series Vol. 523, pp. 124—144.
10. C. Strachey and C. P. Wadsworth, "Continuations: A Mathematical Semantics for Handling Full Jumps," *Higher-order and Symbolic Computation*, Vol(13), pp.135-152, 2000.

## Appendix

Pref ::= sin|cos|sqrt|abs|sum|product|transpose  
Inf ::= +|-|\*|/|^|mod|...  
Rel ::= <|>|<=|>=|==|<>  
Cond ::= T Rel T | not Cond | Cond or Cond | Cond and Cond  
T ::= Id T\* | const | (T\*) | Pref T | T Inf T | T when Cond else T  
S ::= s | (S)  
Func ::= Id : S\* → S | Id(Id\*) ::= T  
Prog ::= Func\*