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 allows a programmer to describe functions in terms of abstract relationships between their inputs and outputs, and 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 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 illustrates this kind of guidance with a simple example.

1 Introduction and Related Work

SequenceL is a 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 parallel algorithms.

The beginning point of this effort can be traced to [1], which overlaps the iterative morphisms found in [6]. 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. The NTD semantic achieves a goal similar to that of the Lämmel and Peyton-Jones, boilerplate elimination. [4,5]

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 arguments, which are *overtyped* (i.e., nesting level higher than expected), the remaining arguments are *normalized* -- duplicated to match the length of the overttyped arguments.

For 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 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.

3 The Environment and Matrix Multiply

From the standpoint of specification, the i,j th element of the product matrix of a and b is the dot product of the i 'th row of a with the j 'th column of b . Here is SequenceL code for the algorithm:

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

This function appears to compute only a single row of the matrix product AB , but will compute all rows if a matrix is passed as an argument, due to NTD. 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 rather than a vector as expected, `normalize` and `transpose` are performed yielding

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

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

which are evaluated (potentially in parallel) for the final result:

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

The parallelisms discovered between step 5 and the final result are micro-parallelisms, and therefore, should be carried out iteratively. As it turns out, steps 4 and 5 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 5 of the Trace

Step 5 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 4 of the Trace

Step 4 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.

4 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, Fast Fourier Transform, 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. 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.
2. 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.
3. 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.
4. R. Lämmel and S. Peyton-Jones, "Scrap your boilerplate: a practical design pattern for generic programming," in Proceedings of TLDI 2003, ACM Press.
5. R. Lämmel and S. Peyton-Jones, "Scrap more boilerplate: reflection, zips, and generalised casts," to appear in Proceedings of ICFP 2004, ACM Press.
6. 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.