SEQUENCE L: TRANSPARENCY AND MULTI-CORE PARALLELISMS

Brad Nemanich  
Texas Tech University  
8th & Boston  
Lubbock, TX 79409-3104  
00 1* 806-742-3527  
brad.nemanich@ttu.edu

Daniel Cooke  
Texas Tech University  
302 Pine Street  
Abilene, Texas 79601  
00 1* 806-742-3527  
dcooke@coe.ttu.edu

J. Nelson Rushton  
Texas Tech University  
8th & Boston  
Lubbock, TX 79409-3104  
00 1* 806-742-3527  
nelson.rushton@gmail.com

ABSTRACT
In this paper, we review the computational laws upon which SequenceL is based, give evidence of the transparency of SequenceL code, and compare performance results of Parallel Haskell and SequenceL codes running on multi-core processors. For the comparisons with Haskell we employ a recently developed SequenceL compiler that generates multi-threaded C++ and has a runtime system that manages threads for multi-core processors.

Categories and Subject Descriptors  
D. Software D.3 Programming Languages D.3.2 Language Classifications: Applicative (functional) languages; D.3.3 Language Constructs and Features: Concurrent programming structures

General Terms  
Performance, Design, Languages, and Theory

Keywords  
SequenceL, Transparency, Computational Laws, Multi-core

1. Introduction  
In this paper we review the computational laws comprising the foundation of SequenceL and give results comparing SequenceL and Parallel Haskell codes running on dual quadcores. The computational laws are formally defined in [CR08]. We will also review the degree of transparency in SequenceL solutions. A more complete paper on SequenceL’s transparency can be found in [CR09]. All example SequenceL codes given in this and all cited papers automatically compile to parallel code, which runs on multi-core processors. The parallel implementations result from a CSP step. For example, given the following tableau:

```
   Consume - Simplify - Produce (CSP) and the Normalize-Transpose (NT). All other features of the language are definable from these laws including recursion, subscripting structures, function references, and evaluation of function bodies. [CR08] Given the fact that the language is strictly based on the CSP-NT, developing assurances concerning the code and handling parallelisms is simplified. Furthermore, SequenceL does not require recursion or specialized operators (like the maps and zips of Haskell) to distribute operations over lists. [HJW92] As a consequence of these properties, SequenceL displays a key feature required for development of trusted software, namely transparency, i.e. that a problem solution can be seen, at a glance, to do what the programmer wants. C.A.R. Hoare long ago expressed the view that “the price of reliability is the pursuit of utmost simplicity.” [Hoare] We claim that compactness, transparency, and verifiability are key to Hoare’s vision of simplicity. In this paper we focus on intuitive descriptions of CSP and NT and the transparency of SequenceL problem solutions. Transparency is illustrated in Section 4 on a variety of simple examples. The problems were chosen because they are (1) well known and/or easily stated, (2) substantially different from each other, and (3) in many cases used as exemplars of transparency in literature on other languages (e.g., primes for APL and quicksort for Haskell). With this hand full of simple examples we hope to make it plausible to the reader that SequenceL facilitates transparency generally. We have tested this hypothesis on medium-scale applications of interest to NASA, and are currently engaged in development of medium-scale commercial applications. Both of these efforts are discussed briefly in Section 6. In section 5 we compare SequenceL to Parallel Haskell in terms of both transparency and performance on multi-core processors.

2. Consume-Simplify-PRODUCE  
A SequenceL interpreter has a workspace called a tableau, consisting of one to many SequenceL terms. The execution of a SequenceL program consists of a series of tableaus each resulting from a CSP step. In general, all terms in a given tableau that reference only constant arguments are consumed. The consumed work is then simplified, and the simplified work is replaced to produce the next tableau. For example, given the following tableau:

```
   Consume - Simplify - Produce (CSP) and the Normalize-Transmute (NT). All other features of the language are definable from these laws including recursion, subscripting structures, function references, and evaluation of function bodies. [CR08] Given the fact that the language is strictly based on the CSP-NT, developing assurances concerning the code and handling parallelisms is simplified. Furthermore, SequenceL does not require recursion or specialized operators (like the maps and zips of Haskell) to distribute operations over lists. [HJW92] As a consequence of these properties, SequenceL displays a key feature required for development of trusted software, namely transparency, i.e. that a problem solution can be seen, at a glance, to do what the programmer wants. C.A.R. Hoare long ago expressed the view that “the price of reliability is the pursuit of utmost simplicity.” [Hoare] We claim that compactness, transparency, and verifiability are key to Hoare’s vision of simplicity. In this paper we focus on intuitive descriptions of CSP and NT and the transparency of SequenceL problem solutions. Transparency is illustrated in Section 4 on a variety of simple examples. The problems were chosen because they are (1) well known and/or easily stated, (2) substantially different from each other, and (3) in many cases used as exemplars of transparency in literature on other languages (e.g., primes for APL and quicksort for Haskell). With this hand full of simple examples we hope to make it plausible to the reader that SequenceL facilitates transparency generally. We have tested this hypothesis on medium-scale applications of interest to NASA, and are currently engaged in development of medium-scale commercial applications. Both of these efforts are discussed briefly in Section 6. In section 5 we compare SequenceL to Parallel Haskell in terms of both transparency and performance on multi-core processors.

2. Consume-Simplify-PRODUCE  
A SequenceL interpreter has a workspace called a tableau, consisting of one to many SequenceL terms. The execution of a SequenceL program consists of a series of tableaus each resulting from a CSP step. In general, all terms in a given tableau that reference only constant arguments are consumed. The consumed work is then simplified, and the simplified work is replaced to produce the next tableau. For example, given the following tableau:

```

SequenceL is a very small, higher order, statically typed, Turing complete, and intuitive language employing strict evaluation. It has 12 grammar rules and very simple semantics. Apart from implementing standard arithmetic, relational, and structural (e.g. list concatenation) operators, SequenceL possesses only two simple computational laws: the Consume-Simplify-
Recursion comes free of charge as a side-effect of the CSP. Consider, for example, the function factorial defined by:

\[
\text{fact}(n(0)) := \text{fact}(n-1) \times n \text{ when } n > 1 \text{ else } 1;
\]

The 0 in fact’s argument indicates that the argument expected is a scalar. Likewise, a 1 would indicate a vector; a 2 would indicate a matrix, etc. Given the initial tableau:

\[
\text{fact}(3)
\]

the CSP will consume the function reference, simplify it by instantiating the variable \( n \) in the function body, and produce the instantiated function body in the next tableau:

\[
(\text{fact}(3-1) \times 3 \text{ when } 3 > 1 \text{ else } 1)
\]

The next CSP evaluates the condition \( 3 > 1 \) to true:

\[
(\text{fact}(3-1) \times 3 \text{ when } \text{true} \text{ else } 1)
\]

Next, the semantics of when are invoked to obtain:

\[
(\text{fact}(3-1) \times 3)
\]

The next CSP evaluates 3-1 and produces:

\[
(\text{fact}(2) \times 3)
\]

The reference to fact(2) is consumed (leaving the \( \times 3 \) in the tableau), the function body is instantiated with 2, and the instantiated function body is produced in the next tableau:

\[
((\text{fact}(2-1) \times 2 \text{ when } 2 > 1 \text{ else } 1) \times 3)
\]

Once again the function body is consumed, the condition is evaluated, and the true expression is left in the subsequent tableau:

\[
((\text{fact}(2-1)) \times 2 \times 3)
\]

The remaining tableaus resulting from subsequent CSP’s are predictable:

\[
((\text{fact}(1)) \times 2 \times 3)
\]

\[
((\text{fact}(1-1) \times 1 \text{ when } 1 > 1 \text{ else } 1) \times 2 \times 3)
\]

\[
((1) \times 2 \times 3)
\]

The need to deploy recursion in SequenceL is significantly reduced when compared to other functional languages. For example, using the so-called “generative construct”, denoted by “...”, the same result is obtained by \( \text{prod}[1,...,3] = \text{prod}[1,2,3] = 6 \), which is similar to a Haskell solution of factorial. However, the Normalize-Transpose operation discussed in the next section further, and more significantly, reduces the need for recursion when compared to other functional languages.

### 3. Normalize-Transpose

The NT often serves as a possible simplification step in the CSP. The basic idea is that functions and operators are defined to work on scalars or nonscalars, which can be nested to different levels: 1, 2, 3, etc. A user-defined function may specify ? to denote the fact that for the associated parameter, any level of nesting is permissible. Having the NT makes it possible to declare only the basic operation to be performed, very often eliminating the need for recursion to break apart nonscalars. If an operator, defined to work on structures of a depth \( n \), is applied to a structure of depth \( m > n \), the applied structure is said to be overtyped and at least one NT is performed. (For example, the function fact, in the previous section is defined to work on structures where \( n = 0 \), i.e., scalars.) More generally, if \( m - n = i \) and \( i > 1 \), NT’s are applied \( i \) times in \( i \) successive CSP’s. If \( m - n = i \) and \( i < 1 \) an error message is issued, and if \( m = n \), no NT is performed. Consider the following examples of the NT at work.

Given a tableau:

\[
[1,2,3] \times 5
\]

The consume will remove the expression from the tableau. However, simplification cannot immediately evaluate the expression since multiplication is defined only on scalars. Consequently, an NT will occur in the simplification step. Since the “overtyped” term \([1,2,3]\) has \( 3 \) elements, three copies of the \( * \) and three copies of the \( 5 \) will be formed by the normalize, resulting in \([[[1,2,3],[*,*,*],[5,5,5]]] \). The transpose (similar to a matrix transpose) completes the simplification step resulting in the production of the next tableau:

\[
[1 * 5, 2 * 5, 3 * 5]
\]

All of these expressions can be evaluated in parallel. So they are all consumed from the tableau, evaluated in the simplification step, and yield the final result:

\[
[5,10,15]
\]

The NT scales naturally. Consider more complicated nested examples:

\[
[[[1,2,3],[4,5,6],[7,8,9]]*2
\]

\[
[[[1,2,3]*2,[4,5,6]*2,[7,8,9]*2]
\]

\[
[[1*2,2*2,2*2],[4*2,5*2,6*2],[7*2,8*2,9*2]]
\]

\[
[[2,4,6],[8,10,12],[14,16,18]]
\]

The middle two tableaus above contain opportunities for parallel evaluation. When an operator has more than one overtyped argument, the NT is performed with respect to the maximally overtyped argument. The NT is performed with respect to the maximally overtyped argument:

\[
[[[1,2,3],[4,5,6],[7,8,9]]*10,20,30]
\]

\[
[[[1,2,3]*10,20,30],[4,5,6]*10,20,30,]
\]

\[
[[7,8,9]*10,20,30]
\]

\[
[[1*10,2*20,3*30],[4*10,5*20,6*30],
\]

\[
[[7*10,8*20,9*30]]
\]

\[
[[10,40,90],[40,100,180],[70,160,270]]
\]

\[
[[10,40,90],[40,100,180],[70,160,270]]
\]

\[
[[10,40,90],[40,100,180],[70,160,270]]
\]

Below is the sequence of tableaus generated when a binary tree of integers is multiplied by a scalar:

\[
[50, [43, [40, 45]], [58, [52, 65]]] * 2
\]
Each relation in the two sets of relations can be evaluated in parallel when \( [false, true, false, true] \)).

The \texttt{when} is handled like any other operator. It will accept any structure on its lefthand side, but accepts only a single Boolean on its righthand side. Therefore, in the next CSP an NT is done with respect to the \texttt{when}:

\[
\text{(word:”a”, count:size([”a” when false, ”b” when false,”c” when false,”b” when false]),}
\text{(word:”b”, count:size([”a” when false,”b” when true,”c” when false,”b” when true]))}
\]

The eight \texttt{when} clauses can be evaluated in parallel. When a \texttt{when} clause has no \texttt{else}, and its condition evaluates to false, an empty is returned:

\[
\text{(word:”a”, count:size([”a”]), (word:”b”, count:size([”b”, ”b”]))}
\]

At this point the size operators can evaluate in parallel:

\[
\text{(word:”a”, count:1), (word:”b”, count:2)}
\]

4. Transparency

There are other languages with similar goals as SequenceL. However, their transparency is often blurred by the need for recursion and/or the need to know specialized operators. For example, an expression for the sum of vectors \( u \) and \( v \) in NESL [B96] is:

\[
(x + y : x \text{ in } u; y \text{ in } v)
\]

and in SequenceL is:

\[
u + v
\]

At this level the NESL and SequenceL syntax are comparably readable, given a small amount of practice in each. However, the NESL comprehension syntax becomes cluttered if we must traverse deeper, nested data structures. Replacing vectors \( u \) and \( v \) with matrices \( a \) and \( b \), in NESL we write:

\[
\{x + y : x \text{ in } u; y \text{ in } v \} : u \text{ in } a; v \text{ in } b
\]

compared with SequenceL’s

\[
a + b
\]

The SequenceL is still readable at a glance. We claim the NESL is not. We do not claim the NESL code is hard to read; a competent NESL programmer can grasp it with only a miniscule probability of error by looking at the code for just a few seconds. But this is typically true of any single line of code in any language. Now make it one of ten thousand lines, and give the programmer the distraction of having to understand the algorithm he is
implementing on top of the code syntax, these miniscule probabilities and few seconds are liable to add up to real errors and real delays. This is why we claim transparency or readability at a glance is important.

In the following sections we provide further evidence of the transparency of SequenceL, when compared to other languages. We highlight the self-claimed best examples of transparency for several languages, including Miranda, APL, ML, and Haskell. In previous papers we have compared SequenceL to other languages, including Gamma [BL93] and FP [B78]. [C96, C97, CA00] The benchmarked problems include finding even numbers, prime numbers, Matrix Multiplication, Jacobi Iteration, and Quicksort. With the exception of SequenceL, not all languages are represented in each example: for instance APL shows up only for the prime number because it is often advertised to be a good example problem for APL. SequenceL versions are given for all of the benchmarked problems and all of the SequenceL versions result in parallelisms in the problem evaluations.

**Even Numbers**

Given a set \( S \) of integers, we define \( \text{evens}(S) \) as the set whose members are the even integers in \( S \). In set-builder, \( \text{evens}(S) = \{ x \mid x \in S \& x \mod 2 = 0 \} \). Replacing the set \( S \) with a list for programming purposes, the most transparent solutions follow:

ML:

\[
\text{fun evens \[] = \[] | evens \(h::t\) = if \(h \mod 2 = 0\) then \(h::\text{evens} \ t\) else \text{evens} \ t;}
\]

Haskell:

\[
\text{evens::[Int] \rightarrow [Int]} \\\\\\\\\\\text{\text{evens} [] = []} \\\\\\\\\\text{\text{evens} \(x::xs\) \\\\\\\\\\\mid \text{even} \(x\) = x:(\text{evens} \ xs) \\\\\\\\\\\mid \text{otherwise} = \text{evens} \ xs}
\]

An alternative Haskell solution is defined by

\[
\text{evens :: [Int] \rightarrow [Int]} \\\\\\\\\\\text{\text{evens xs} = \text{filter even} \ xs}
\]

The ML and Haskell solutions are obscured by the fact that list operators are applied to break apart and reconstruct the lists using recursion—or by both the writer and reader having to know the meaning of ‘filter’. The Miranda and SequenceL solutions do not require recursion or special operators and both are easily mapped to the specification. [Note: The Miranda solution will also work in Haskell. The Haskell solution above is designed to work with infinite lists through lazy evaluation; otherwise one can use list comprehension.]

Miranda:

\[
\text{evens \(e:x\) = e:\{n \mid n \neq x; \ n \mod 2 = 0\}}
\]

SequenceL:

\[
\text{evens(int(0)) := int when (int \ mod 2) = 0;}
\]

**Jacobi Iteration**

Jacobi Iteration for the solution of a partial differential equation is specified by equation:

\[
\mu_{j,k} = \left((\mu_{j-1,k} + \mu_{j+1,k} + \mu_{j,k-1} + \mu_{j,k+1})/4 - ((\mu_{j,k} * \Delta^2)/4)
\]

when neither \( j \) or \( k \) subscript the first or last row or the first or last column otherwise return the border value

The best solutions to this specification (for one iteration) are given Haskell and SequenceL.

Haskell:

\[
\text{jacobi a delta = map } (\text{i} \rightarrow [\text{jacobi_helper a delta i j} | \ j < \[0..(\text{length } a!!i)-1\]]) \\
\quad \quad [0..(\text{length } a)-1]
\]

\[
\text{jacobi_helper a delta i j} = \\
\quad \quad \text{\mid i} == 0 \mid j == 0 \mid (\text{length } a)-1 \text{ \mid (length } (a!!i))-1 \text{ \mid j = a!!i!!j} \\
\quad \quad \text{\mid otherwise} = \\
\quad \quad \text{\quad (a!!(i+1))!!j + a!!(i-1)!!j + a!!!!(j+1) + a!!!!(j-1))/4 - ((a!!!!j) * delta^2)/4)
\]

Notice, that the jacobi helper is close to the specified equation. However, a second order operator, the map, is used to distribute the jacobi_helper over the elements of the matrix. Therefore, one must know how this specialized operator works and work through the \( ij \) manipulations to be certain that indeed the two functions implement the specified equation. This is a small issue for the programmer, but a larger issue for non-programmers who might want to inspect the code, such as managers, customers, and domain engineers.

SequenceL:

\[
\text{\text{jacobi}( a(2),b(2), \delta(0))}[j,k] := \\
\quad \quad (a!!i+k)!!j + a!!(j-l)!!k + a!!!!(j+1)!!k + a!!!!(j-1)!!k)/(4- \\
\quad \quad \text{\quad ((b[j,k] * \delta^2))/4})
\]

\[
\text{\quad \quad \text{\quad when not((j = 1 or size(a)=j) or)} \\
\quad \quad \text{\quad \quad \text{\quad (k=1or size(a)=k))}
\]

\[
\text{\quad \quad a[j,k];}
\]

Notice all of the operators are intuitive and no specialized functions need to be known so that the SequenceL function aligns well to the specifying equation.

**Prime Numbers**

A set builder expression for the set of prime numbers in the set \( S \) of integers is: \( \{ x \mid x \in S \& (x > 1 \rightarrow \forall \{i \in [2,\ldots,\sqrt{x}] \mid (x \mod i \neq 0) \text{ OR } x = 2 \} \} \)

For the special case where \( S = \{1,\ldots,R\} \), a commonly touted APL expression is given by:
The APL definition is terse and not transparent. Specialized knowledge of operators is needed. It is so difficult to read that an in-depth explanation is warranted.

APL is right associative. The \( \iota \) generates the numbers from 1 to the limit \( R \). If \( R = 6 \), then the list is \( 1,2,3,4,5,6 \). The down-arrow on the list strips off the 1 and the left arrow assigns the resulting vector to \( R \). From there, \( R^\circ \times R \) generates the outer product of the vector, which presents a matrix of the values obtained by multiplying the vector times itself:

<table>
<thead>
<tr>
<th>x</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
</tr>
</tbody>
</table>

Next, using set membership an APL selection vector is constructed. Each element of the selection vector indicates whether a corresponding number in \( R \) is in the table. The vector produced in our example in which \( R \) is 6 is \( (0,0,1,0,1) \). The selection vector is negated and then, using the / operator, the corresponding elements from the original vector are selected:

\[
(1,1,0,1,0)/(2,3,4,5,6) \text{ yields } (2,3,5).
\]

A Miranda function for the general case is:

```miranda
countPrimes :: [Int] -> [Int]
countPrimes [] = []
countPrimes (a:x) = countPrimes [b | b < a; b <= a] ++ [a] ++
countPrimes [b | b < a; b > a]
```

And finally the equivalent in SequenceL:

```sequencel
countPrimes(x(1)) := x when (all(x mod (2...ceiling(sqrt(x))) /= 0) or x=2) and x>1;
```

SequenceL maps well to the set builder definition. Notice it is not a recursive solution and like the other functions result in parallelisms.

**Quick-sort**

Quick-sort is an efficient sorting technique. It is a naturally recursive definition, which follows:

- Pick one item from the array--call it the pivot
- Partition the items in the array around the pivot so all elements to the left are smaller than the pivot and all elements to the right are greater than the pivot
- Use recursion to sort the two partitions

Haskell:

```haskell
quickSort :: [Int] -> [Int]
quickSort [] = []
quickSort (s:xs) = quickSort [x|x <= xs, x < s] ++ [s] ++ quickSort [x|x <= xs, x >= s]
```

Miranda:

```miranda
countPrimes :: [Int] -> [Int]
countPrimes [] = []
countPrimes (a:x) = countPrimes [b | b < a; b <= a] ++ [a] ++
countPrimes [b | b < a; b > a]
```

SequenceL:

```sequencel
countPrimes(x(1)) := x when (all(x mod (2...ceiling(sqrt(x))) /= 0) or x=2) and x>1;
```

All three of the definitions are effectively the same and align well with the specification.

**Matrix Multiplication**

In Matrix Multiplication, the \((i,j)\)'th element of the solution is the sum of the corresponding dot products of the \(i\)th row of the first matrix and the \(j\)th column of the second matrix. More formally:

\[ (AB)_{ij} = \sum_{r=1}^{n} A_{ir} B_{rj} \]

Haskell:

```haskell
matMul :: [[Int]] -> [[Int]]
matMul a b = [[dotProd row col | col <- transpose b] | row <- a]
dotProd :: [Int] -> [Int] -> Int
dotProd x y = sum [s * t | (s,t) <- zip x y]
```

We argue that this solution is less transparent than the SequenceL version that follows, not just because it is longer, but also because one must know what the `zip` in the `dotProd` function does and the fact that \( b \) must be transposed in the `matMul` function.

SequenceL:

```sequencel
countPrimes(x(1)) := x when (all(x mod (2...ceiling(sqrt(x))) /= 0) or x=2) and x>1;
```

The `all` indicates all elements of the corresponding row or column.
5. Experiments

This section presents comparison data from experiments focused on the respective speeds of SequenceL and parallel Haskell on three problems: matrix multiplication, word search, and quicksort.

The data we present in this paper uses the Haskell compiler GHC version 6.10.1 running on Xenon Dual Quad Core Processors. The SequenceL compiler is written in a sequential version of Haskell and generates multi-threaded C++ code for the same machine. Both languages have a runtime component for the multi-core processors. For each experiment we ran 20 trials on each configuration (i.e., 20 on 1 processor, 20 on 2 processors, etc.).

The matrix multiplication was performed in both languages on a 1000 × 1000 matrix. We found the parallel Haskell version on a Haskell website (www.haskell.org). Below is the parallel Haskell version of the matrix multiplication problem:

```
multMat :: [[Int]] -> [[Int]] -> [[Int]]
multMatT :: [[Int]] -> [[Int]] -> [[Int]]
multMatT m1 m2T = [[multVec row col | col < m2T] | row < m1]
multVec :: [Int] -> [Int] -> Int
multVec v1 v2 = sum (zipWith (*) v1 v2)
multMatPar :: Int -> [[Int]] -> [[Int]]
multMatPar z m1 m2 = (multMat m1 m2) `using` strat z
strat = blockStrat
lineStrat c = parListChunk c rnf
(blockStrat c matrix -- best?
    = let blocks = concat (splitIntoClusters numB matrix) -- result
        splitted
            -- in numB * numB blocks
        numB = round (sqrt (fromIntegral (length matrix) / fromIntegral c))
            -- approx. same num/granularity of sparks as
            in others...
            in parList rnf block

        type Vector = [Int]
type Matrix = [Vector]

splitIntoClusters :: Int -> Matrix -> [[Matrix]]
splitIntoClusters c m | c < 1 = splitIntoClusters 1 m
splitIntoClusters c m1 = mss
    where bh = kPartition (length m1) c
        bhsplit [] [] = []
bhsplit [] _ = error
            "some elements left over"
bhsplit (t:ts) xs = hs : (bhsplit ts rest)
    where (hs,rest) = splitAt t xs
        ms = bhsplit bh m1 -- blocks of rows
        mss = map (colsplit bh) ms
        colsplit [t] rs
            | head rs == [] = []
            | otherwise =
                (cab:colsplit ts rest)
                where (cab,resto) = unzip (map (splitAt t) rs)

-- helper for splitIntoClusters (formerly
bresenham)
kPartition :: Int -> Int -> [Int]
kPartition n k = zipWith (+)
    ((replicate (n `mod` k) 1) ++ repeat 0)
    (replicate k (n `div` k))
```

The parallel code in SequenceL is the same as was seen previously:

```
matmul(x(2),y(2))[i,j] :=
    sum( x[i,all] * y[all,j]);
```

Note the transparency in the SequenceL code, and how transparency is extended by the fact that the code requires no annotation to guide the parallelisms. There are no such annotations needed or available in SequenceL. The comparative speeds on the same 1000 × 1000 matrix are shown below:

![Matrix Multiply Performance](image)

The x-axis is clock speed and the y-axis is the number of processors – speedups are implied by the graph.

We were unable to find parallel Haskell codes for grep. So our best Haskell programmer wrote the Parallel Haskell version of a simple Grep (i.e., no regular expressions). We mention this, because we expect that there are better performing versions in parallel Haskell. We experimented with adding par and seq commands to different parts of the program and show the results for the version with the best performance. The grep was performed on a 10,000,000 character file searching for a 5 character word. The parallel Haskell version of the simple – word search – grep is:

```
grep :: [String] -> String -> [String]
grep lines key = filter (substring key) lines
substring :: String -> String -> Bool
substring [] = True
substring _ [] = False
substring (x:xs) (y:ys) = checkFront `par`
    (checkRest `pseq` (checkFront || checkRest))
    where
        checkFront = isPrefix (x:xs) (y:ys)
        checkRest = substring (x:xs) ys
```
isPrefix :: String -> String -> Bool
isPrefix [] _ = True
isPrefix _ [] = False
isPrefix (x:xs) (y:ys) = (x==y) && (isPrefix xs ys)

We found this solution to be easier to read and it contains only one ‘par’ annotation to direct parallelisms. In SequenceL, the same problem is solved by:

grep(a(1),b(1)) :=
word_search(a,b,1...(size(a)-size(b)+1));
word_search(a(1),b(1),n) :=
let str := a[n...(n+size(b)-1)];in
str when eq_list(str,b);

The SequenceL Quicksort follows:
great(a,b):=a when (a>b);
less(a,b):=a when (a<b);
quick(a(l)):=
(quick(less(a,a[1]))++a[1]++quick(great(a,a[1]))) when (size(a)) > 1 else a;

We found this solution to be easier to read and it contains only one ‘par’ annotation to direct parallelisms. In SequenceL, the same problem is solved by:

grep(a(1),b(1)) :=
word_search(a,b,1...(size(a)-size(b)+1));
word_search(a(1),b(1),n) :=
let str := a[n...(n+size(b)-1)];in
str when eq_list(str,b);

The Quick sort is an interesting parallel problem, because parallelisms are dynamic (i.e., you cannot predict parallelisms before execution because of the pivot). The experiment involved a list of 5,000,000 integers. We found the parallel Haskell version on a Haskell website: (http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/sec-gph.html). Here is the parallel Haskell version of Quicksort:
quicksortS [] = []
quicksortS [x] = [x]
quicksortS (x:xs) = losort ++ (x:hisort)
`using` strategy
where
losort = quicksortS [y | y<- xs, y<x]
hisort = quicksortS [y|y<-xs, y>= x]
strategy result =
  rnf losort `par`
rnf hisort `par`

rnf result `par` ()

The SequenceL Quick sort follows:
great(a,b):=a when (a>b);
less(a,b):=a when (a<b);
quick(a(l)):=
(quick(less(a,a[1]))++a[1]++quick(great(a,a[1]))) when (size(a)) > 1 else a;

Notice again that the SequenceL programmer does not have to identify the parallelisms. The performance results are:

![Performance of QuickSort on 5,000,000 Integers](image)

6. Past, Current, and Future Work

Up until recently we have had interpreters, but no parallel compilers for SequenceL. The SequenceL compiler is less than a year old. We have identified a number of high payoff optimizations we can perform on the compiler and on the runtime system. We anticipate that clockspeeds and speedups will improve dramatically once these optimizations are in place.

During the past several years we have been investigating the transparency of SequenceL definitions for other formal, non-numerical problems, including relational database primitives, set builder notation, and denotational semantics. We have an expanding number of example cases that indicate the transparency of SequenceL when transcribing definitions from these areas. For example, the denotational semantic equations of a Turing Complete while-language are, with minor changes, executable in SequenceL.

With NASA collaboration we have also worked with success on medium sized Guidance, Navigation, and Control (GNC) systems that are based partially on Natural Language requirements. [CBLG06] Our most recent success is a significant application for the Space Shuttle, called the Predictor Code (PEG for short). During the launch phase the predictor code estimates every two seconds where the vehicle will be two seconds later. The computations involve thruster information and aerodynamic calculations that determine among other things, the altitude and momentum of the spacecraft so that abort options can be determined quickly should the Main Engines Cut Off.
PEG was written in the 1970’s in FORTRAN and rewritten in C in the 1980’s. There is a 90 page requirements document associated with PEG. GNC engineers and Lockheed are considering reusing PEG for the Orion Crew Exploration Vehicle (CEV). They know the code works – it is a trusted system, but they need to know exactly what it does so that they can align PEG with CEV requirements.

The SequenceL group developed a 25 page PEG requirements document for NASA. In November, 2007 we presented the document to a group of GNC engineers. One of the engineers had been working to understand the PEG requirements and code with little success. Seeing our requirements, the engineer said he could easily read the document and understand the system. We pointed out that the document is a commented SequenceL program and is consequently executable. We add this as anecdotal evidence on the transparency of SequenceL.

We have since tested the SequenceL version of PEG, which worked correctly after remedying minor errors – mainly typo’s. It is worth noting that those typo’s might likely go undetected in a static requirements document.

Currently we are working with a third party software company that is implementing, in SequenceL, applications for a major controls processing company. For one of the applications, the goal of the company was to obtain a 12x improvement over their C-Code version of the application. The compiled SequenceL code obtained a 27x improvement on 8 cores. The SequenceL version of the test case took less than an hour to write.

7. Conclusion
In this paper, we reviewed the CSP and NT semantics and provided examples of the transparency of SequenceL when compared to APL, Miranda, and Haskell. The transparency is a subjective decision; programmers with significant experience with the other languages may disagree with our analysis. However, it has been our experience that scientists and engineers with no experience with SequenceL can immediately understand SequenceL declarations of problem solutions in their respective domains.

The transparency of SequenceL extends to parallel solutions. The SequenceL programmer does not annotate SequenceL code for parallelisms. Finally we also provided more objective evidence of the relative parallel performance of SequenceL and parallel Haskell on multi-core processors.

8. ACKNOWLEDGMENTS
Our thanks to NASA: NNG05GP48G, NNG06GJ14G, and NNG06GJ14G.

9. REFERENCES