Automatic Concurrency in SequenceL

Daniel E. Cooke [a]

[a] *Department of Computer Science*
Texas Tech University
Lubbock, TX 79409, U.S.A.
email dcooke@coe.ttu.edu

Vladik Kreinovich [b]

[b] *Department of Computer Science*
University of Texas at El Paso
El Paso, TX 79968, U.S.A.
email vladik@cs.utep.edu

Abstract

This paper presents a programming language which we believe to be most appropriate for the automation of parallel data processing, especially data processing of concern to the oil industry and to the U.S. Federal Agencies involved in the analysis of Satellite Telemetry Data. Focus is placed upon major language issues facing the development of the information power grid. The paper presents an example of the type of parallelism desired in the Grid. To implement this parallelism in such a language as Java we need to specify parallelism explicitly. We show that if we rewrite the same solution in the high level language SequenceL, then parallelism becomes implicit. SequenceL seems therefore to be a good candidate for a Grid Oriented Language, because its abstraction relieves the problem solver of much of the burden normally required in development of parallel problem solutions.

Key words: Automatic concurrency, SequenceL, term-rewriting systems

1 The Need for New Language Abstractions

Hardware improvements and the general spread of computing and computer applications have created opportunities for scientists and engineers to solve ever more complicated problems. However, there are concerns about whether scientists and engineers possess the software tools necessary to solve these problems and what computer scientists can do to help the situation.

The fundamental software tool for problem solving is the programming language. A programming language provides the abstraction employed in solving problems. In order to keep pace with hardware improvements, computer scientists should continually address the problem of language abstraction improvement. When advances in hardware make problems *technically feasible* to solve, there should be corresponding language abstraction improvements to make problems *humanly feasible* to solve.

In the recent past, most language studies have resulted in the addition of new features to existing language abstractions. The most significant changes have resulted in additions to language facilities for the definition of program and data structures. These changes have primarily taken place to accommodate the needs for concurrent execution and software reuse. Although it is important to *add to the existing abstractions* to satisfy immediate technical problems, research also needs to be undertaken to *simplify and minimize existing abstractions*.

There are application domains where the need for simpler language abstractions is of vital importance. There are estimates that less than 1% of the available satellite data has been analyzed. There exists the ability to acquire and store the data, but weakness in the ability to determine its information content. Soon NASA will have satellites in place that, in sum, will produce a Terabyte of data per day. A major problem associated with the analysis of the data sets is the time needed to write the medium-to-small programs to explore the data for segments containing information pertinent to particular earth science problems. Software productivity gains in developing exploratory programs are needed in order to enhance the abilities of earth scientists in their efforts to grapple with the complexity and enormity of satellite and seismic data sets. Software productivity gains can be accrued through languages developed out of foundational research focusing on language design.

The need for computer language abstraction improvement is even more pronounced given the desire to develop distributed approaches to data analysis. Currently, industry and government agencies are paying a lot of attention to approaches involving complicated data parallel solutions. Data parallelisms embody the idea of *scatter/gather* approaches to problem solving, where data is *scattered* among several different processors which process the corresponding pieces of the original data set, and then the results of this processing are assembled (*gathered*) together to produce the final result. Most such parallelizations use Single-Instruction-Multiple-Data (SIMD-) type architecture where a single program executes on multiple, networked processors. This "scatter/gather" approach to computing has been very successful, e.g., in the analysis of seismic data sets.

Prior to the SIMD approach, the oil industry would analyze entire seismic data sets on a single "super computer." The SIMD approach was adopted by many companies in the early 1990's and has since resulted in cheaper and faster processing of seismic data sets. These data sets are used to determine which sites companies should lease for their offshore drilling activities. The seismic data sets (upon which scatter/gather approaches have proven to be successful) have quite a bit in common with the satellite telemetry data sets that NASA and other federal agencies acquire and store. There is a major effort to generalize the SIMD architecture by developing a super system that could employ idle resources on the World Wide Web. The effort is generally called the *Information Power Grid*, or the *Grid* for short.

The Information Power Grid is a major effort funded by a number of U.S. federal agencies including NASA and the NSF. The goal of this effort is to establish a computing infrastructure on the world wide web, providing powerful supercomputing level resources to any user connected to the web. "The grid will connect multiple regional and national computational grids to create a universal source of computing power. The word 'grid' is chosen by analogy to the electric power grid, which provides pervasive access to power..." [4].

One way to envision the goal of this effort is to imagine a web browser button that would allow the user to submit programs for execution. In an ideal case, the program would be analyzed to determine the parallelisms it contains. Then, a suitable distributed, parallel architecture would be configured by seizing idle processors connected to the Internet – the envisioned system would provide to all entities connected to the web, access to teraflop computing capabilities. Clearly there are a number of technical challenges that face those who are developing the grid. The focus here is on the computer language issues.

"Powerful new strategies for supporting the development of high-performance distributed applications will be needed... The application developer should be able to concentrate on problem analysis and decomposition at a fairly high level of abstraction... To do this, [the programming support system] will need to find every possible type of parallelism within the application, including data parallelism and task or object parallelism... From the user's perspective, the most appealing approach to program decomposition is automatic parallelism." [5]

In this paper, we will focus on language solutions to the programming support system referred to in the preceding passage. We will first show a simple data parallel problem solution using Java's multithreading features. We will then describe a very high level language, SequenceL, and indicate how the same data parallel problem solution is easily identifiable in the SequenceL solutions. One goal of the paper is to convince the reader that SequenceL holds promise

as a grid-oriented language.

2 Data Parallelisms in Java

The key to achieving high performance on distributed-memory machines is to allocate data to various processor memories to maximize locality and minimize communication [5]. Data parallelism is parallelism that derives from subdividing the data domain in some manner and assigning the subdomains to different processors. Data parallelisms (e.g., those characteristic of SIMD-type architectures) typically result in the same computation being performed simultaneously on subdivided data sets, as opposed to dividing up the computation itself.

As an example, we will consider a word search problem: to find all occurrences of a desired word s_1 of length n_1 in a given string s of a larger length $n > n_1$. We will illustrate this problem on the example of searching for the word `test` of length $n_1 = 4$ in a string `here is a test string` of length $n = 21$. In principle, the tested word can start in any of the positions from 0 to $n - n_1$ of the longer string. Therefore, a straightforward parallelizable algorithm for solving this problem consists of checking, for each such place i, whether a substring of s of length n_1 starting at this place coincides with s_1. The corresponding sequential Java program is as follows:

```
String s="here is a test string";
String s1="test";
char[]sample=s.toCharArray();
char[]find=s1.toCharArray();

System.out.println(sample);
n=sample.length;
n1=find.length;

for(i=0;i<=n-n1;i++)
            {System.out.println}(s.substring(i,i+n1));
             if(s.substring(i,i+n1).equals(s1))
               {System.out.println}(i);}
            }
```

This algorithm can be naturally parallelized: if we have sufficiently many processors, we can ask different processors to check the equality of substrings corresponding to different starting places i. However, even in Java, a language specifically designed for computation over the Web, this natural parallelization is not so easy to describe. The resulting code is given in the Appendix.

This solution uses a built-in construction `thread` which describes parallelizable threads of a computation process. In this solution, an array `w` consisting of $n-n_1+1$ (=18 in our example) substring variables is declared (in line 33) and filled with the corresponding substrings (lines 35–38). This "filling" initializes the 18 instances of the class constructor method `wrdsrch2` (lines 7–12). Once the 18 instances are set up, the processes of comparing the strings are initiated and executed concurrently (in lines 42–43). When these 18 processes end, they join into the main process, and the 18 instances of the boolean variable `found` are then printed as output.

Even when we know the sequential program, the concurrent solution to this problem is not easy to write and not easy to understand. It uses difficult-to-understand special language constructs such as `thread`, `try`, `join`, `run`. The next sections of the paper are intended to convince the reader that the high level executable language SequenceL may provide a more suitable abstraction for representing data parallelisms.

3 Introducing the SequenceL Language

SequenceL was introduced as an approach to software development that offers a different, and for many, a more intuitive approach to problem solving [2,3]. For an exact description of SequenceL, the reader is referred to [2,3]. We will just mention that there exists a rather efficient interpreter for this language, and a new, even more efficient interpreter is being completed. SequenceL is *universal* in the usual sense: the universal Turing machine can be described in this language and therefore, an arbitrary algorithm can be described in it. In this paper, we briefly (and informally) describe the basic ideas behind SequenceL, the basic constructions, and how they help in parallelization.

The main idea underlying the design of SequenceL is the idea – similar to declarative languages – that ideally, the main product of the software developer should be the exact description of *what* the program should achieve and not necessarily *how* to achieve it. In traditional languages, programmers write explicit algorithms; these algorithms implicitly contain all the relations between the input data and the output of the program that we want to implement by writing this program. The goal of the SequenceL design effort is to provide a language in which a programmer would, instead, explicitly formulate the exact relationship between the input and the output, and then the compiler will choose an appropriate algorithm depending on such factors as the availability of parallelization.

Consider as an example a simple program to compute the mean of several (n) data values. In the traditional approach one states an algorithm (i.e., a

step-by-step sequence of instructions) that will produce the desired result:

Traditional Approach - Pseudo Code

1. *Get the numbers, one at a time, counting them as they are read.*
2. *Add the values together (sum them).*
3. *Divide the sum by the count obtained in Step* 1.

In SequenceL, one explicitly declares the desired result:

SequenceL Approach - Pseudo Code

The desired output is the ratio of the sum of the input values and the number of the input values.

This reformulation would help to overcome one of the main difficulties of traditional programming that drastically impedes its productivity – the difficulty of understanding what exactly is computed by a given program. Complexity of a program is caused by the complexity of its *data structures* and especially by the complexity of its *control structures*. Software engineers have long realized that the construction of loops is complex and costly [6]. Bishop noted that "Since Pratt's paper on the design of loop control structures was published more than a decade ago, there has been continued interest in the need to provide better language features for iteration" [1].

To avoid the complexity of *data structures*, SequenceL has only one *data type construction*: a *list* (sequence) $[s_1, \ldots, s_n]$. By using this list construction, we may go from basic data constants (also called *singletons* or *scalars*) to non-scalar types: lists of singletons and nested lists (lists of lists). Whenever this does not lead to confusion, singletons are identified with one-element lists. Nested structures can be nested to any depth. In other words, a *constant* is a term build from singletons by using a sequence construction $[s_1, \ldots, s_n]$.

We can also allow variables as singletons. For the resulting more general terms, it makes sense to allow the notation $s(i)$, meaning i-th element of the list s. As we will see in the following text, we will sometimes need to interpret the expression $s(i)$ for values i which are larger than the number of elements in s. We will use the following interpretation of $s(i)$ for such i: we repeat the list s again and again until we reach i, so, e.g., $[10, 30, 50](4) = 10$, $[10, 30, 50](5) = 30$, etc. For lists of lists, we can similarly define $s(i, j)$ as $s(i)(j)$, i.e., as j-th element of the list $s(i)$.

To avoid the complexity of *control structures*, SequenceL defines a *program* also as a sequence, namely, as a sequence consisting of lists and function symbols.

Roughly speaking, a SequenceL program consists of:

(a) data which can be viewed as sequences of symbols, and
(b) rules which define substitutions of strings by other strings.

The execution of a program consist in applying rules to the data. (This ideology is somewhat similar to that of term rewriting systems but rules can be substantially more complex.)

Function symbols can be of four types:

- *Binary* symbols correspond to functions of two variables and are described in infix notation, like + in $2+3$; the left argument will be called a *predecessor* of the binary function symbol, and the right argument will be called its *successor*.
- There are also two types of unary symbols, corresponding to *postfix* notation (like factorial ! in $n!$) and *prefix* notation (like sin in $\sin(x)$).
- We can also have functions without inputs.

Functions $f(x_1, \ldots, x_n)$ of three or more variables are described as functions of a single variable – namely, of a list $[x_1, \ldots, x_n]$.

There is only one type of control operation: built-in recursion, in which a subsequence of a program which contains a function symbol is replaced by a new subsequence which describes the result of the corresponding function. The original subsequence is said to be *consumed*, and the new replacement is said to be *produced*.

- The replacement result may be a constant, e.g., `2+2` is replaced by `4`.
- This result can itself contain a function symbol, e.g., a factorial expression `fact[n]` is replaced by `n*fact[n-1]` when `n>1` and by `1` otherwise.

There are three different types of basic functions:

- The most basic type includes *regular* operations which operate on all elements of the operand list; e.g., a (binary) *addition* operator $a+b$ adds corresponding elements of the two lists a and b, while the unary *sum* operator $+a$ adds all the elements of a list a. Thus, $+[5] = [5]$, $+[4,4,3,2] = [13]$, and the sum $+[[10,20,30,40,50],[4,5,6,7,8]]$ is defined as $[10,20,30,40,50] + [4,5,6,7,8]$, i.e., as a component-wise sum $[14,25,36,47,58]$. If different lists contain different number of elements, we *normalize* them by repeating elements of the smaller list again and again: e.g.,
$$+[[10,20,30,40,50],[4,5,6]] = +[[10,20,30,40,50],[4,5,6,4,5]] = [14,25,36,44,55].$$
- In contrast to regular operations which are applied to *all* elements of the list, *irregular* operations are only applied to those elements which satisfy

a certain condition. For example, if the list `salary` contains salaries of all the faculty, and the list `evaluation` contains their evaluations, then the conditional unary multiplication operation

```
*[salary(i),1.1] when evaluation(i)>5
```

means that we increase by 10% the salary of all the faculty whose evaluations are better than 5.

- There are also *generative* constructions which describe standard shorthand ("three dot") notations, e.g., `[1,...,5]` is interpreted as the list $[1, 2, 3, 4, 5]$.

More complex functions can be defined by combining the basic functions. For example, if we describe a matrix a as a list of its rows

$$[[a_{11}, \ldots, a_{1n}], \ldots, [a_{m1}, \ldots, a_{mn}]],$$

so that $a(i, j)$ is exactly a_{ij}, then we can define a binary operation of matrix multiplication as follows:

```
Function matmul(consume(pred(n,*),succ(*,m)),produce(next)),
  where next(i,j)=+[pred(i,*)*succ(*,j)]
  taking (i,j) from [1,...,n] X [1,...,m]
```

The intent of this description is that if the function symbol `matmul` appears in the program between the lists representing two matrices, say, `a` and `b`, and if we apply this function, then the substring

```
a matmul b
```

is replaced by a single list which represent the product of the two matrices `a` and `b`. Let us describe how this intent is reflected in the above SequenceL description.

- The word `Function` is a standard term of SequenceL, and the following word `matmul` is the name of the newly defined function.
- The information in parentheses which immediately follows the word `consume` describes the input to the function `matmul` (i.e., describes what is "consumed" by this function):
  - The fact that this information contains both the words `pred` (*predecessor*) and `succ` (*successor*) means that thus defined function is a binary function in infix notation. In other words, the symbol `matmul` must appear in a program in between two lists, a *predecessor* list `pred` and a *successor* list `succ`.
  - The appearance of two indices in `pred` (namely, the expression `pred(n,*)`) means that `pred` is a list of lists (i.e., crudely speaking, a matrix). The first index `n` describes the number of elements in the list `pred`. The number of elements in the each of `n` sublists is denoted by a wild-card symbol `*`, which

means that it must be the same for all these sublists. In our explanation, we will denote this common number by `p`.

  - The appearance of two indices in `succ` (namely, the expression `succ(*,m)`) means that *succ* is also a list of lists (matrix). The first index `*` is the same wild card as for `pred`, which means that the number of lists (rows) in `succ` must be the same as the number of elements `p` in each of the sublists of `pred` (i.e., in each column of `pred`).

- The information in parentheses which immediately follows the word `produce` describes the output to the function `matmul` (i.e., describes what is "produced" by this function). This output is a list called `next`.
- The information after the word `where` is a body of the function it describes what the output `next` looks like:
  - The appearance of two indices in `next` (namely, the expression `next(i,j)`) means that `next` is also a list of lists (matrix).
  - The expression

    `next(i,j)=+[pred(i,*)*succ(*,j)]`

    describes the value `next(i,j)` for all possible `i` and `j`:
    - the wild card symbol `*` in the expression `pred(i,*)*succ(*,j)` means that we consider the same value of the corresponding index, i.e., we consider the products `pred(i,k)*succ(k,j)]` for different values of the wild card index `k`;
    - the infix multiplication symbol in `[prev(i,k)*succ(k,j)]` means a componentwise multiplication of the lists

      `[prev(i,1),...,prev(i,p)]`

      and

      `[succ(1,j),...,succ(p,j)]`;

      in other words, we create a list of products

      `[prev(i,1)*succ(1,j),...,prev(i,p)*succ(p,j)]`.
    - finally, the `+` in front of the list means that this `+` is the above-described unary sum operation, which adds all the elements of the above list of products:

      `next(i,j)=prev(i,1)*succ(1,j)+...+prev(i,p)*succ(p,j)`.

    The line

    `taking (i,j) from [1,...,n] X [1,...,m]`

    means that we take all pairs of indices `(i,j)` for which `i=1,...,n` and `j=1,...,m`.

So, this function indeed describes the desired matrix multiplication.

Comment. To be more precise,

`[1,...,n] X [1,...,m]`

indicates a *lexicographically ordered* Cartesian product, i.e., the (ordered) set of all possible pairs of indices

$$\{(1,1),(1,2),\ldots,(1,m),(2,1),\ldots,(2,m),\ldots,(n,1),\ldots,(n,m)\}.$$

4 SequenceL's Computational Model

As we have mentioned, the execution of a program in SequenceL is similar to a term rewriting system: a subterm of a certain type is replaced by a different subterm, etc., until the further reduction is impossible. However, SequenceL is more general than usual term rewriting systems:

- in a term rewriting system, the replacing term is, in essence, a combinatorial transformation of the original terms (permutations, repetitions, deletions, etc.), while
- in SequenceL, the replaced term can be obtained from the original term by an arbitrary algorithm.

Let us illustrate this idea in more formal terms, on the simplified case of programs which contain no variables. Let S be a set of all possible sequences obtained from basic constants by using the list operation [.,...,.] and the index operation (.).

Let N be the set of all function symbols. As we have mentioned earlier, a program is a finite sequence consisting of elements of S and function symbols. The set of all the programs will be denoted by Π. For each function symbol $f \in N$, we define its *type* $H(f)$:

- for binary functions in infix notation (which have both predecessor and successor), the type is defined as a set $\{pred, succ\}$;
- for prefix unary functions, the type is $\{succ\}$;
- for postfix unary functions, the type is $\{pred\}$; and
- for functions without inputs, the type is the empty set $\{\}$.

In other words, the set of all possible function types is $D = 2^{\{pred,succ\}}$, and H is a function from the set N (of all function symbols) to D.

To describe the meaning of a function symbol $f \in N$, we must describe how a subsequence containing f (and no other function symbols) is replaced by a new subsequence. Depending on the function type, the original subsequence is of one of the types f, αf, $f\beta$, or $\alpha f \beta$, where α and β are lists from S. The set of all possible subsequences of these types can be described as $F = N \cup (S \times N) \cup (N \times S) \cup (S \times N \times S)$. Thus, the *meaning* B of different function symbols is defined as a (partially defined) function which maps subsequences into new subsequences, i.e., as a partially defined function from F to Π.

If we add a special symbol *undefined* whenever the function symbol is not defined, then we can describe the meaning as a total (everywhere defined) function $B : F \to \Pi \cup \{undefined\}$. This function B must be consistent with the type $H(f)$ of each function symbol f: e.g., if f is a binary function symbol,

then $B(f)$ can only be defined for triples (α, f, β) and undefined for elements of $N \cup (S \times N) \cup (N \times S)$.

For example, if we allow natural numbers and parentheses as basic constants and arithmetic operations as function symbols (with standard interpretation), then the expression $(4+5)/(5-2)$ is an example of a program. A subsequence $4+5$ corresponds to a triple (α, f, β), with $\alpha = 4$, $f = +$, and $\beta = 5$. The meaning $B(4, +, 5)$ of this subsequence is the number 9. For a subsequence ")/(", the meaning is undefined.

We say that a substring δ of a program P is *enabled* if the "meaning" function B is defined for this substring. The set of all enabled substrings of a program P will be denoted by *Enabled*(P). For example, the above program $P = (4+5)/(5-2)$ has two enabled substrings: $4+5$ and $5-2$, so *Enabled*$(P) = \{4+5, 5-2\}$.

Now, we can describe how a SequenceL program is executed. An execution of a program consists of a sequence of steps. On each step, one or several disjoint enabled substrings δ_i are replaced by their meanings $B(\delta_i)$. Formally, for each program P for which *Enabled*$(P) \neq \emptyset$, we define *Execute*(P) as the set of all sequences $\gamma_1 B(\delta_1)\gamma_1 B(\delta_2) \ldots \gamma_n B(\delta_n)\gamma_{n+1}$, for which P can be represented as $P = \gamma_1\delta_1\gamma_2\delta_2 \ldots \gamma_n\delta_n\gamma_{n+1}$ for some substrings $\gamma_1, \delta_1, \ldots, \delta_n, \gamma_{n+1}$ $(n > 0)$.

For example, since the program $P = (4+5)/(5-2)$ contains two disjoint enabled substrings, *Execute*$(P) = \{P', P'', P'''\}$, where:

$$P' = (B(4, +, 5))/(5-2) = (9)/(5-2),$$
$$P'' = (4+5)/(B(5, -, 2)) = (4+5)/(3),$$
$$P''' = (B(4, +, 5))/(B(5, -, 2)) = (9)/(3).$$

A *computation* of a program P is then defined as a sequence $P_1, \ldots, P_n$, in which $P_1 = P$, $P_{i+1} \in$ *Execute*(P_i), and *Enabled*$(P_n) = \emptyset$.

In our example, computations in which $P_2 = P'$ or $P_2 = P''$ correspond to *sequential* computations in which only one arithmetic operations is performed at a time. Computation in which $P_2 = P'''$ correspond to the *concurrent* solution, in which both addition and subtraction are computed on the same computation step. This concurrent solution is represented by a computation sequence $P_1 = (4+5)/(5-2)$, $P_2 = (9)/(3)$, and $P_3 = 3$.

5 Data Parallelisms in SequenceL

We have seen that the computation model of SequenceL naturally leads to concurrency. Let us now show how a similar concurrency naturally emerges in the above word search problem.

In SequenceL terms, the above word search algorithm can be described by the following function:

```
Function search(consume(pred(n),succ(n1)),produce(next)),
  where next(x)=(pred(x)=succ)
  taking x from [[1,...,n1],...,[n-n1+1,...,n]]
```

This description means that the symbol `search` can appear in a program only between two lists, a predecessor list `pred` ("long text") whole length is denoted by `n`, and a successor list `succ` ("short text") whose length is denoted by `n1`. As a result of the function `search`, the two lists `pred` and `succ` are replaced by a single list `next`. This list consists of `n-n1+1` Boolean values `next(x)`; each value is equal to `true` or `false` depending on whether `pred(x)=succ`, i.e., whether the short text `succ` of length `n1` is indeed contained in the long list at `n1` consecutive indices `x` (=`[1,...,n1],[2,...,n1+1],...,[n-n1+1,...,n]`).

In particular, in our example, when `search` is applied to the texts

`prec=[here is a test string]` and `succ=[test]`,

the index `x` takes 18 possible values:

```
[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7],
[5,6,7,8],[6,7,8,9],[7,8,9,10],[8,9,10,11],
[9,10,11,12],[10,11,12,13],[11,12,13,14],[12,13,14,15],
[13,14,15,16],[14,15,16,17],[15,16,17,18],[16,17,18,19],
[17,18,19,20],[18,19,20,21]
```

The function `search` replaces both strings `pred` and `succ` with a single list of 18 Boolean values of the following 18 relations:

```
[[here]=[test],[ere ]=[test],[re i]=[test],[e is]=[test],
[ is ]=[test],[is a]=[test],[s a ]=[test],[ a t]=[test],
[a te]=[test],[ tes]=[test],[test]=[test],[est ]=[test],
[st s]=[test],[t st]=[test],[ str]=[test],[stri]=[test],
[trin]=[test],[ring]=[test]]
```

Now, we need to find the truth values of all these 18 relations. In view of the above-described computational model of SequenceL, it is clear that all 18 values can be computed concurrently, resulting in the following list:

```
[false,false,false,false,
 false,false,false,false,
  false,false,true,false,
 false,false,false,false,
       false,false]
```

In essence, we have the exact same natural parallelization as in the Java program presented in the Appendix: the `taking` construction subdivides the larger data set into 18 smaller sets just like like the Java program does in lines 35–38 and 7–12. However, the parallelisms in SequenceL are much more intuitive: in SequenceL, parallelization naturally comes from the program itself, and, in contrast to Java, this parallelization does not require changing the program or using any additional constructions like `thread`, `run`, etc.

To further test the parallelization abilities of SequenceL, we are currently designing an efficient parallel interpreter for this language.

6 Conclusions

SequenceL seems to provide a more intuitive approach to data analysis problems – especially when parallelisms are required in the solution. Even the most modern computing languages (e.g., Java) are somewhat cumbersome when it comes to the design and understanding of parallel solutions. Modern approaches to data analysis as exemplified by the goals of the Grid project require languages that can express parallelisms at a higher level – languages for which parallelisms can be identified automatically.

SequenceL is presented as a candidate Grid Oriented Language. SequenceL is a high level universal language that provides an abstraction suitable for automatically generating iterative and parallel program structures. The language is based upon a simple execution strategy similar to term rewriting systems. We believe that this language is a good candidate for a Grid Oriented Language – a language appropriate for describing and using high parallelism of potential Grid applications.

Although the example data parallel problem solution developed in this paper is rather simple, the example scales up to many real-world data mining problems involving image processing and security-based data searches.

Acknowledgments

This work was supported in part by NASA under cooperative agreement NCC5-209 and grant NCC 2-1232, by NSF grants No. DUE-9750858, CDA-9522207, and 9710940 Mexico/Conacyt, by the United Space Alliance, grant No. NAS 9-20000 (PWO C0C67713A6), by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants F49620-95-1-0518 and F49620-00-1-0365, by the National Security Agency under Grants No. MDA904-98-1-0564 and MDA904-98-1-0564, and by Grant No. W-00016 from the U.S.-Czech Science and Technology Joint Fund

The authors are thankful to the anonymous referees for helpful suggestions, and to Michael Gelfond for valuable discussions.

References

[1] J. Bishop, *The Effect of Data Abstraction on Loop Programming Techniques*, IEEE Trans. Soft. Eng. **SE-16** (1990), 389–402.

[2] D. Cooke, *An Introduction to SequenceL: A Language to Experiment with Nonscalar Constructs*, Software Practice and Experience **26** (1996), 1205–1246.

[3] D. Cooke, *SequenceL Provides a Different way to View Programming*, Computer Languages **24** (1998), 1–32.

[4] I. Foster and C. Kesselman, *Preface* to: I. Foster and C. Kesselman (eds.), "The Grid, Blueprint for a New Computing Infrastructure", Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[5] K. Kennedy, *Compilers, Languages, and Libraries*, in: I. Foster and C. Kesselman (eds.), "The Grid, Blueprint for a New Computing Infrastructure", Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[6] H. Mills and R. Linger, *Data Structured Programming: Programming without Arrays and Pointers*, IEEE Trans. Soft. Eng. **SE-12** (1986), 192–197.

Appendix

```
class wrdsrch2 extends Thread{ String text; String target;
boolean found; int i;

wrdsrch2(String in, String targ, int k) {           Line  7
    target=targ;                                    Line  8
    text=in;                                        Line  9
    found=false;                                    Line 10
    i=k;                                            Line 11
}                                                   Line 12

public void run()   {
    if(text.equals(target))
        {found = true;}
}

public static void main (String args[]) {
    int i, j, k, n, n1;

    String s = "here is a test string";
    String s1 = "test";
    char[] sample = s.toCharArray();
    char[] find = s1.toCharArray();

    System.out.println(sample);

    n  = sample.length;
    n1 = find.length;
    String send;

    wrdsrch2 w[] = new wrdsrch2[(n-n1)+1];          Line 33

    for(i=0;i<=n-n1;i++)                            Line 35
        {send = s.substring(i,i+n1);                Line 36
         w[i] = new wrdsrch2(send,s1,i);            Line 37
        }                                           Line 38

    System.out.println("To Run ");

    for(i=0;i<=n-n1;i++)                            Line 42
        {w[i].start();}                             Line 43
```

```
    for(i=0;i<=n-n1;i++)                                    Line 45
        {try {w[i].join();                                  Line 46
         catch (InterruptedException ignored) { }           Line 47
            }                                               Line 48

    System.out.println("The answer is:  ");

    for(i=0;i<=n-n1;i++)
        {System.out.println(w[i].found);}

}}
```