Fully Automatic Parallel Programming

Bryant Nelson and Nelson Rushton(contact author)
(bryant.nelson | nelson.rushton) @ ttu.edu
Dept. of Computer Science, Texas Tech University
Box 43104 Lubbock, TX 79409-3104

Abstract - *SequenceL is a small, statically typed, purely functional programming language, whose semantics enable compilation to parallel executables from function definitions. This paper reports the results of experiments on the performance of parallel programs automatically generated by the SequenceL compiler. In particular we examine the parallel speedups obtained in running SequenceL programs on multicore hardware.*

Keywords: SequenceL, Functional Programming, Parallel Programming

1 SequenceL

SequenceL is a simple, general purpose, purely functional programming language [Cooke et.al. 2008]. By *simple*, we mean that the entire syntax and semantics of the language can be described in about 20 pages, including examples. By *general purpose*, we mean the language is not specific to any domain; and by *purely functional* we mean that SequenceL programs consist of equations defining functions, without any I/O or assignment. To be part of a working executable program, SequenceL programs are compiled to C++ and linked with so-called "driver code" that orchestrates I/O operations.

The original aim of SequenceL was to give programmers a way to describe computations in terms of the relation between input and output data, without direct reference to a particular procedure for obtaining them [Cooke et. al. 2009]. On its surface, this sounds like the aim of functional languages in general, but in reality almost all functional languages act as shorthand for known procedures. For example, the author of a Haskell or Lisp program does not necessarily have to think about how his program will be executed (say, left to right lazy evaluation, or left to right eager evaluation, respectively), but if he *does* think about it, he may know exactly how it will execute because the language semantics make guarantees about the order. The semantics of SequenceL make no such guarantees.

Because SequenceL makes no guarantees about the order of evaluation, it is not possible for a SequenceL programmer to optimize their code in a compiler-independent way. On the other hand, this means the compiler may perform optimizations in ways that are not constrained as in other languages. In particular, since the language makes no guarantees about the order of evaluation, evaluations may be done in parallel. SequenceL's Normalize-Transpose semantic (see Section 2) is particularly amenable to parallelisms being automatically discovered and exploited by the compiler. This automated parallelism was first pointed out in [Cooke/Andersen 2000], and implemented as a prototype with encouraging results reported by Nemanich [Nemanich et. al. 2010]. In 2009 the patent on SequenceL's semantics was licensed to Texas Multicore Technologies (TMT), who have since been engaged in commercial scale development of the compiler.

2 Parallelizations by SequenceL

2.1 Normalize-Transpose

The parameters of a SequenceL function are explicitly typed according to their *depth*. *Depth* can be thought of as the dimensionality of an expression. For example, scalars have depth 0, lists have depth 1, matrices have depth 2, etc. One way in which SequenceL alleviates the need for iterative or recursive algorithms is with Normalize-Transpose (NT). NT is a method of function application that applies some operation on every element in a list. A function defined on

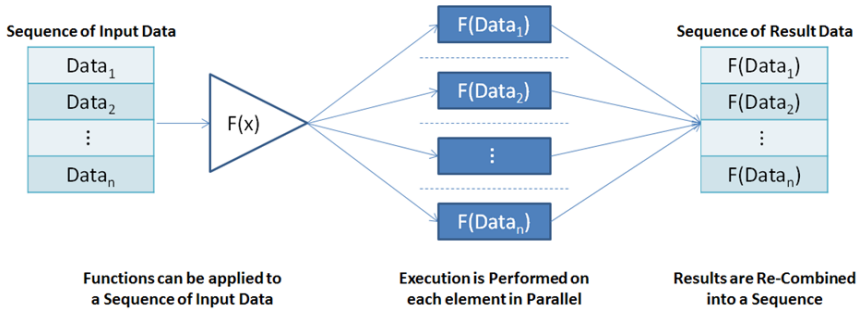

Figure 1: NT Illustration

arguments of depth D can be applied to a list of arguments of depth D. The result is the function applied element-wise.

For example, since the scalar addition function (+) is declared with scalars (depth 0) for both arguments, the expression [10,20,30] + 1, where the first argument is a list, triggers an NT and has a value of [11,21,31]. Similarly because of NT, the value of the SequenceL expression [10,20,30] + [1,2,3] is [11,22,33]. The NT semantic is one device that allows SequenceL to automatically extract parallelizations. It can be proven that the parallelisms generated are free of race conditions and other parallel anomalies.

2.2 Indexed Functions

Another way SequenceL avoids the use of recursion is through a construct called indexed functions. Using indexed functions a programmer can specify a nonscalar data structure element-wise, a function of the parameters of the function. This is very similar to the way vector and matrix valued functions are often defined in informal mathematics. Take for example the *Identity* function defined below -- where for each nonnegative integer *N*, *Identity(N)* is the *NxN* identity matrix:

```
Identity(N(0))[i, j] :=
    1 when i = j
  else
    0
  foreach i within 1 ... N,
          j within 1 ... N;
```

Figure 2: Identity Matrix as Indexed Function

2.3 Consume-Simplify-Produce

The third source of automatic parallelizations in SequenceL is that parameters of a function call may be evaluated in parallel. This is known as Consume-Simplify-Produce, or CSP.

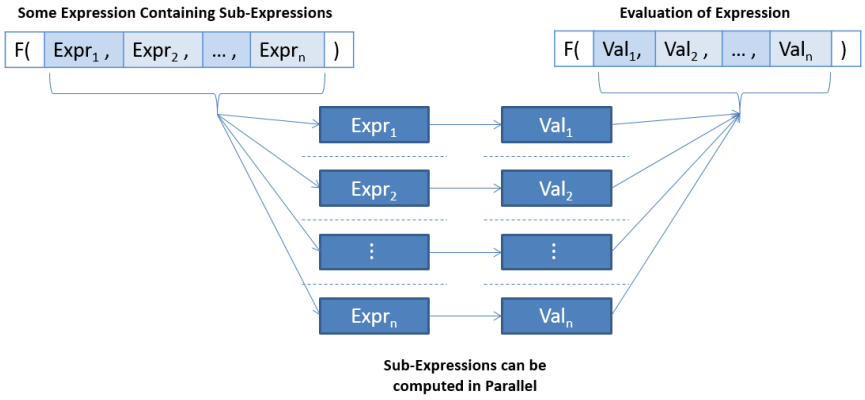

Figure 3: CSP Illustration

When SequenceL is compiled to C++ CSPs, indexed functions, and NTs are compiled into highly parallel programs, capable of running on an arbitrary number of processor cores. The number of cores can be specified at runtime.

3 Heat Map and its Explanation

A set of benchmarks informally known as a heat map is periodically run to test the performance of compiled SequenceL. The heat map problems have been chosen essentially at random from modules that have been written for commercial TMT customers over the past three years. Tables 1 and 2 list all of the heat map problems that are at least 70 lines of SequenceL code. The cut-off at 70 lines of code was chosen to represent problems that cannot be trivially parallelized by hand.

The problems in Table 1 (LU factorization, 2D Fourier Transform, and the Barnes-Hut N-body problem) have well known specifications and can be considered repeatable experiments, which can be used for performance comparison between SequenceL and other methods including by-hand parallelization. The problems in Table 2 (Semblance, Compare Predicates, Speech Filter, and WirelessHART) are not repeatable in the sense that they contain algorithms proprietary to TMT customers. They are listed here as anecdotal observations.

The heat map reports the average run time over 10 executions of several different programs and compares the performance of the SequenceL on 1, 2, 4, 8, 12, 16, 20 and 24 cores. The following table shows the average run times over 10 executions when run on a Centos 6.3 machine with 16GB of memory, and a 1333MHz / E5-2620, running at 2.0 GHz.

Table 1: Standard Algorithms

Cores	LU Factorization	2D Fourier Transform	Barnes-Hut N-Body
1	12.426	2.690	29.850
2	6.335	1.347	15.976
4	4.130	0.674	8.830
8	2.973	0.338	4.765
16	2.494	0.245	3.483
12	2.400	0.218	3.297
20	2.425	0.215	3.174
24	2.511	0.203	3.007

Table 2: Proprietary Algorithms

Cores	Semblance	Compare Predicates	Speech Filter	WirelessHART
1	20.993	3.341	108.742	21.996
2	11.335	2.167	54.364	18.000
4	5.962	1.129	29.324	14.668
8	4.038	0.654	20.021	13.194
12	3.100	0.593	14.353	12.993
16	2.542	0.608	16.185	13.196
20	2.902	0.607	11.142	13.484
24	3.371	0.559	9.979	13.770

4 Conclusions and Future Work

The SequenceL compiler generated parallel algorithms automatically, without human intervention between the functional description of the solution and the parallel executable. Parallel speedups were obtained in every case. In most cases the speedups continued nearly linearly up to around 8 cores. Above 8 cores, performance either increased slightly or decreased slightly as more cores were added, with the exception of *speech filter* in which substantial speedups were obtained up to 24 cores.

This "core ceiling" phenomenon for linear speedups is not unexpected in general, since any parallel program running on a physical machine will eventually reach such a threshold due both to communication overhead and to the theoretical limits of parallelization for the algorithm. The fact that the threshold (of 8 cores) was consistent across problems indicates that in this case the ceiling may have been hardware dependent. This is especially plausible here, because the machine used in this experiment has only 12 distinct physical processors, with up to 24 simulated through hyper-threading.

In our experience this performance is competitive with the performance of hand coded parallel algorithms -- though of course the reader with similar or greater experience may draw their own conclusions. Future work includes a comparison of this performance with hand coded parallel implementations of the same algorithms, comparison with the performance of hand coded sequential algorithms, and running the experiment on different hardware.

5 References

[Cooke/Andersen 2000] Daniel E. Cooke, Per Andersen: Automatic parallel control structures in SequenceL. Softw., Pract. Exper. 30(14): 1541-1570 (2000)

[Cooke et.al. 2008] Daniel E. Cooke, J. Nelson Rushton, Brad Nemanich, Robert G. Watson, Per Andersen: Normalize, transpose, and distribute: An automatic approach for handling nonscalars. ACM Trans. Program. Lang. Syst. 30(2) (2008)

[Cooke et. al. 2009] Daniel E. Cooke, J. Nelson Rushton: Taking Parnas's Principles to the Next Level: Declarative Language Design. IEEE Computer 42(9): 56-63 (2009)

[Nemanich et. al. 2010] Brad Nemanich, Daniel E. Cooke, J. Nelson Rushton: SequenceL: transparency and multi-core parallelisms. DAMP 2010: 45-52
[1] 06).