SURVEY OF PARALLEL PROGRAMMING LANGUAGE FEATURES

A Project Report

Presented to

Thomas Austin

Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Class

CS 252

By

Tapomay Dey

May 2016

ABSTRACT

Parallel programming is gaining importance due to increase in complexity of typical systems and volume of data flowing into such systems. However, most contemporary programmers are trained with the mindset of legacy programming paradigms and constructs. These constructs fall short when it comes to building faster solutions that leverage the full potential of modern hardware. This report is a survey of programming language features that facilitate parallel implementations. It focusses on parallel programming features in functional languages. These features allow a programming style that involves use of highly composable parts. It also highlights the difference in mindset required for writing parallel code.

Contents

1 INTRODUCTION

Reduction in computation time using simple techniques has always been a focus of many fields of Computer Science and Engineering. Compilers perform code optimization for better performance and correctness. Chip manufacturers improve clock speeds and memory access time for faster computation. Similarly, programming languages offer parallel and concurrent programming features that scale-up computation speed by optimal use of available hardware resources. Ease of use and adaptation by programmers has been an important success metric for such programming models.

Sequential and parallel code have different implementation concerns. One can optimize sequential code by minimizing number of operations. Clever algorithm design techniques like dynamic programming re-use pre-computed results among overlapping subproblems. Parallel code often involves reduction in communication and synchronization by performing redundant operations. Sequential code algorithms optimize space usage but parallel algorithms often require extra space. Sequential code decomposes problems in a linear manner. Parallel code uses multiway decomposition. Thus, writing parallel code requires a different mindset [8].

Parallel programming is considered hard. There are two significant reasons for this. It is a nightmare to debug errors in parallel code. Bugs are often not reproducible. A bug may appear and disappear without consistency. Also, different execution environments may lead to different logical behavior. Difference in logical flow may still be a correct execution. But it

makes reasoning about correctness or performance of the system very difficult. Secondly, a parallel implementation of an algorithm must dedicate a large fraction of the code to communication and synchronization concerns. This reduces readability of code because the core algorithm is hidden amidst all the plumbing code. One can minimize the inconvenience caused by these problems by writing code that works on both sequential and parallel environments. We can debug such code in a sequential environment for finding logical errors. Then run the patched code in a parallel computer and get the same results. Programming languages must provide special parallel programming constructs for facilitating this approach [3].

Another inspiration for building better parallel programming models is the death of Moore's law. It has guided the silicon chip manufacturing industry for decades. It was proposed by Intel co-founder Gordon Moore in 1965. However, in 2016, it is believed that Moore's law, as we know today, is finally dead [1]. This means that processor speeds may not increase in the same way as before.

The report starts by showing the basic parallel programming constructs in Haskell in section 2. Section 3 shows an example where a monad operator is inherent to data parallelism in a trivial problem. Section 4 shows how algorithm design techniques map to parallel programming models. Section 5 is a list of design concerns while implementing parallel programming models in a language.

2 PARALLEL PROGRAMMING IN HASKELL

2.1 WHY HASKELL?

This report focusses on Haskell's parallel programming features. The following justifications are based on common knowledge. Haskell is a fertile ground for implementing abstractions and parallel programming is only one such abstraction. Haskell is a purely functional language. Functions can't have any side-effects. All side effects are segregated in an IO Monad. IO Monad may use pure functions but not vice versa. Haskell is strongly typed and it has type inferencing. It also uses lazy evaluation. There are no objects in Haskell. It uses algebraic datatypes and pattern matching. It is easy to reason about the logic and correctness of a code in a language with such features [5]. Any two subexpressions of a data partition are evaluated as pure functions and thus can be evaluated in parallel [3].

2.2 SEQUENTIAL CODE

Consider a basic operation:

```
1  solve :: String -> Maybe Result
```

The function 'solve' takes a String as input and produces a type Result as output. The idea is to take an array of String inputs and parallelize the invocations of the solve function one each array element.

```
1  main = do
2  [ f ] <- getArgs
3  inputs <- fmap lines $ readFile f
4  mapM_ ( evaluate . solve ) inputs
```

The basic sequential solution reads input strings from a file and maps over each input String, invoking solve to get a Maybe Result type. The evaluate function returns the result in an IO monad thus forcing evaluation.

2.3 EVAL MONAD

Definition of the Eval monad:

```
1  data Eval a
2  instance Monad Eval
3  runEval :: Eval a -> a
4  rpar :: a -> Eval a
5  rseq :: a -> Eval a
```

Parallel solution using Eval Monad:

```
1  let ( as , bs ) = splitAt ( length inputs `div` 2)  inputs
2  evaluate $ runEval $ do
3      a <- rpar ( deep ( map solve as ) )
4      b <- rpar ( deep ( map solve bs ) )
5      rseq a
6      rseq b
7      return ()
```

This solution divides the input into two halves and parallely invokes solve on elements of each half using map function. It uses rseq to wait for the parallel invocations to return. The Eval monad uses two basic operations rpar and rseq to synchronize by declaring order of evaluations. rpar is a combinator that creates parallelism and rseq forces sequential evaluation.

2.4 STRATEGIES

We can now define rpar and rseq in terms of Strategy

```
type Strategy a = a -> Eval a
rseq :: Strategy a
rpar :: Strategy a
```

A Strategy is just a function that yields a computation in the Eval Monad. Strategies typically use the ‘using‘ operator:

```
using :: a -> Strategy a -> a
x ‘using‘ s = runEval ( s x )
```

Using Strategies we can express a parallel mapping function:

```
parMap f xs = map f xs ‘using‘ parList rseq
```

Here, parList function is a Strategy on lists. It is a parametrized Strategy and it represents a family of strategies that evaluate list elements in parallel:

```
parList :: Strategy a -> Strategy [ a ]
parList strat [] = return []
parList strat ( x : xs ) = do
    x’ <- rpar ( x ‘using‘ strat )
    xs’ <- parList strat xs
    return (x’: xs’)
```

2.5 PAR MONAD

ParMonad allows specification of dependencies and task boundaries. Although it is a concurrency construct, it also guarantees determinism. Thus it bridges the gap between parallel and concurrent Haskell.

```
newtype Par a
```

```
    instance Functor Par
    instance Applicative Par
    instance Monad Par
runPar :: Par a -> a
fork :: Par () -> Par ()
```

The fork function spawns a new child that executes concurrently with the current execution.

3 AUTO-PARALLELISM

This sections describes a representative problem and the potential for a parallel solution to it. Consider the problem of adding N numbers. We know that plus operator is a monad because we can define the following operations:

- An identity function that adds zero to its input.
- A function that takes two Numbers and adds them to produce a Number.
- And we also know that addition is associative.

```
addZero(x) = x
Add :: Number -> Number -> Number
Add (Add x y) z == Add x (Add y z)
```

We may define a sequential solution as follows:

```
ACC = 0
FOR I = 1 ...N:
  ACC = ACC + A[I]
END FOR
```

```
5   RETURN ACC
```

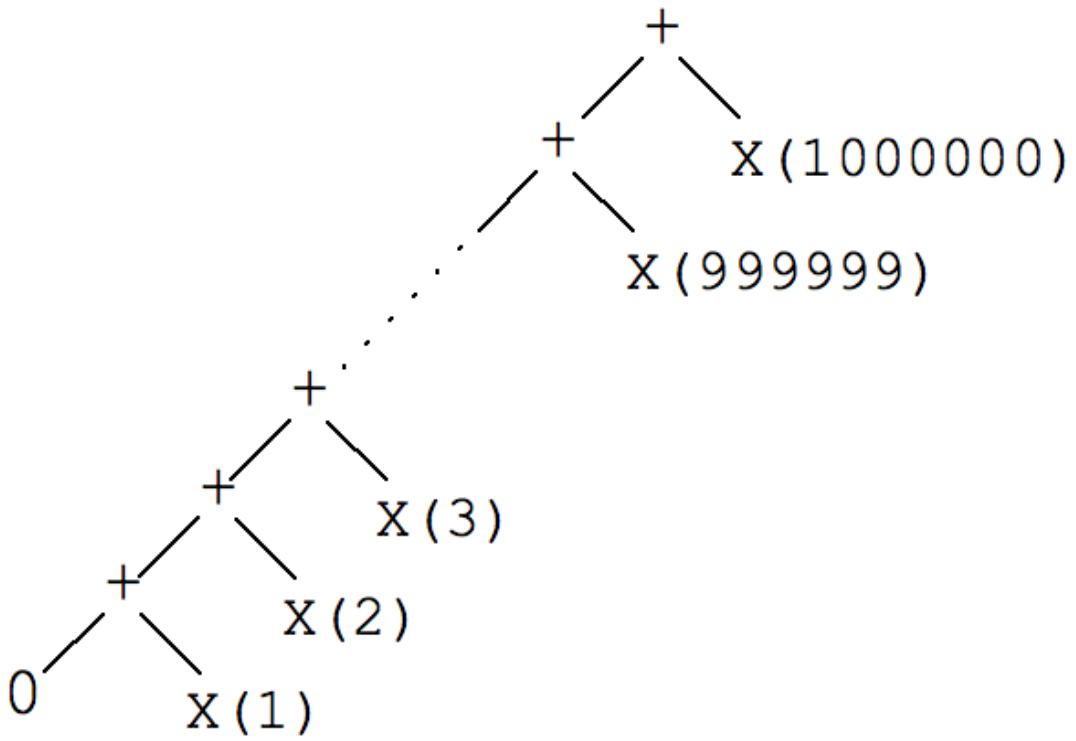

Figure 1: Computation tree of sequential solution. Source: [8]

The left skew in the computation tree is indicative of a sequential computation. A parallel solution should typically lead to a balanced binary computation tree.

The addition monad that we defined above is suitable as the + operator for such an implementation. Any code written using such operator would have the following advantages. User may define code using the operator without any concern of parallelizing it explicitly. Depending on implementation of the operator, it may be transformed to a sequential or parallel implementation by the compiler.

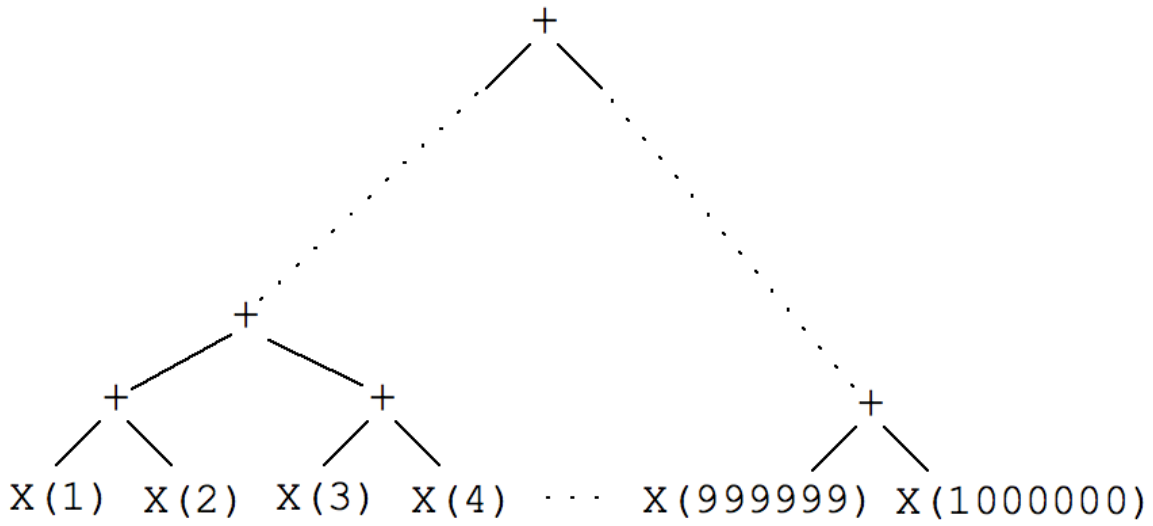

Figure 2: Computation tree of parallel solution. Source: [8]

4 ALGORITHM DESIGN TECHNIQUES AND DATA PARALLELISM

4.1 DIVIDE AND CONQUER

A divide and conquer solution to a problem typically comprises of recursive application of three steps Divide-Conquer-Merge. Such a recursive approach is typically represented using a recurrence relationship [2]. If a binary relation is applicable to a problem then the recurrence relation is bijective to a complete binary tree similar to figure 2.

Typically, a recursive solution has a base case that is applicable to problems that are small enough to compute and end the recursion. This is representative of an identity operator similar to addZero defined in section 3. If we define a associative monad operator that implements the merge operation then such a solution is a suitable candidate for parallelisation by compiler

optimizations similar to the one discussed in section 3.

4.2 DYNAMIC PROGRAMMING

The key ingredients that make a problem suitable for applying dynamic programming are as follows [2] :

- Optimal substructure : Optimal solution to subproblems can be re-used and merged into an optimal solution to the super problem. This is typically visible as a recurrence relation.
- Overlapping subproblems : The recursive solution to the main problem solves the same subproblems over and over again.
- Memoization : The solutions to the subproblems can be mapped and stored for re-use.

A problem that qualifies for dynamic programming can be typically solved using three incrementally optimal approaches [2]:

- Top down recursive implementation
- Top down with memoization using dynamic programming
- Bottom up method using dynamic programming

In the bottom up approach, smaller sub-problems are computed before solving the larger sub-problems that depend on the smaller ones. This is typically represented using a sub-problem graph that represents these dependencies using edges. Such a representation is convenient for estimating the running time of an algorithm using the number of edges.

The sub-problem graph representation is also suitable for writing a parallel solution using constructs like Par Monad as discussed in section 2.5. The edges of a sub-problem graph map directly to dependencies and task boundaries.

5 LANGUAGE FEATURE DESIGN CONCERNS

Following are some fundamental aspects that affect the design of a language feature in terms of parallelism:

- Strictness: Strictness is desirable. In a non-strict language, runtime analysis is required to detect the level of potential parallelism that can be achieved.

- Purity: Side effects introduce indirect dependencies that restrict the degree of potential parallelism. Thus pure functions are desirable.

- Level of Control: Indicative of the degree of details about parallelism control expected from a programmer. The extremes are fully implicit versus fully explicitly with different degrees of declarativeness in between.

- Type System: Distinction between types specific to sequential and parallel solutions may add convenience to the language constructs. Strongly typed languages guarantee correctness and this adds to convenience when debugging and reasoning about a code.

6 CONCLUSION

We saw that a different mindset can greatly influence the design of code that uses highly composable parts and leaves the decision of using parallel or sequential evaluation to a smart compiler or runtime. Such an approach makes it convenient to reason about parallel code and debug errors. Algorithm design decisions like multiway decomposition and monad operator definitions are intrinsic to parallel evaluations. It is easier to define such constructs in functional languages.

References

[1] Bright, P. (2016, Feb 10). *Moore's law really is dead this time. Arstechnica. Retrieved from* `http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/`

[2] Cormen, T. H. (2009). *Introduction to algorithms. MIT press.*

[3] Hammond, K., & Michaelson, G. (Eds.). (2012). *Research directions in parallel functional programming. Springer Science & Business Media.*

[4] Jones, M. P., & Hudak, P. (1993). *Implicit and explicit parallel programming in Haskell. Disponível por FTP em nebula. systemsz. cs. yale. edu/pub/yale-fp/reports/RR-982. ps. Z (julho de 1999). [Online]. Available:* `http://cs-www.cs.yale.edu/publications/techreports/tr982.pdf`

[5] Marlow, S. (2012). *Parallel and concurrent programming in Haskell. In Central European Functional Programming School (pp. 339-401). Springer Berlin Heidelberg. [Online]. Available:* http://community.haskell.org/~simonmar/par-tutorial-cadarache.pdf

[6] Marlow, S., Newton, R., & Peyton Jones, S. (2011, September). *A monad for deterministic parallelism. In ACM SIGPLAN Notices (Vol. 46, No. 12, pp. 71-82). ACM. [Online]. Available:* http://community.haskell.org/~simonmar/papers/monad-par.pdf

[7] Obradovic, D. (1998). *Structuring functional programs by using monads. [Online]. Available:* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.3974&rep=rep1&type=pdf

[8] Steele, G. (2009). *The Future is Parallel: What's a Programmer to do? Guy Steele's 2009 lecture on parallelism at M.I.T. [Online]. Available:* https://groups.csail.mit.edu/mac/users/gjs/6.945/readings/MITApril2009Steele.pdf