PARALLEL PROGRAMMING USING FUNCTIONAL PROGRAMMING LANGUAGE

Vandana Sunag ¹, Sameeksha Irkal ²

¹Lecturer, Bachelor of Computer Applications, K.L.E’S B.C.A, P.C.Jabins College, Vidyanagar, Hubballi, Karnataka, India
²Lecturer, Bachelor of Computer Applications, K.L.E’S B.C.A, P.C.Jabins College, Vidyanagar, Hubballi, Karnataka, India

ABSTRACT:

Multiprocessors systems in the computer field are the current trend to speed up the processing. In order to support this hardware, we need parallelism of data as well as tasks. In this paper, we have tried to cover parallelism through the pure functional language. In functional languages, there are no looping statements like ‘for’, ‘while’ & ‘goto’, only functions are used. Calling the same function twice with the same parameters will result into the same output. And this gives the added benefit that there are no side effects of function. Haskell is one of the popular pure functional languages. Glasgow Haskell Compiler (GHC), is an open source native code compiler for the functional programming language Haskell. It provides a cross-platform environment for the writing and testing of Haskell code.

Keywords: Data parallelism, Pure Functional Language, Haskell, Glasgow Haskell Compiler (GHC)

INTRODUCTION

Increasing the computer speed has researched the peak point now, so we are offering computers containing more and more CPUs. As the number of CPUs increases, it becomes more and more difficult for a programmer to interactions of large numbers of threads. In the world of massively-parallel computing, Data parallel computation is the best methods used in today’s super computers.

Kinds of parallelis
Data parallelism

The same task runs on different data in parallel. Dividing the data among multiple processors is a method of getting data parallelism. For Example: convert all characters in an array to upper-case. This process will divide parts of the data between different tasks and perform the tasks parallel.

![Data parallelism](image1)

1) Task parallelism

Different tasks running on the same data. Example: Several functions on the same piece of data-average, minimum, binary or, geometric mean. No dependencies between the tasks, so all can run in parallel.

![Task parallelism](image2)

2) Pipeline parallelism

Pipeline parallelism is used when multiple steps depend on each other, but the execution can overlap and the output of one step is streamed as input to the next step. Each task can run in parallel. The purpose of pipeline parallelism is to increase your speed of your program and decrease your I/O operations.

![Pipeline parallelism](image3)
Parallel Programming is Difficult

Parallel Processing Imperative programs have side effects. Variables are assigned values and these variables may be accessed by several processes. This introduces the risk of race conditions. A Race Condition Example:
Two processes P1 and P2, which both can write a shared variable tmp:
P1:    P2:
      .  tmp = 4711
      .   z = tmp + 3
    tmp = 17
      .   y = 2*tmp

The intention of the programmer is to both P1 and P2 should set tmp and then use its newly set value right away. But the programmer missed that the processes have a race condition for tmp, since they may run at different speed. Here is one possible situation:

<table>
<thead>
<tr>
<th>P1:</th>
<th>P2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp = 17</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>tmp = 4711</td>
</tr>
<tr>
<td>y = 2*tmp</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>z = tmp + 3</td>
</tr>
</tbody>
</table>

Final state: y = 9422, z = 4714

Pure Functional Programs Language

Functional programming solves computational problems by the chained evaluation of functions, whose output is determined by their inputs that is by its arguments. In this style of programming, side effects and mutable data are restricted. In purely functional programming you don't tell the computer what to do as such but rather you tell it what stuff is. The factorial of a number is the product of all the numbers from 1 to that number, the sum of a list of numbers is the first number plus the sum of all the other numbers, and so on. You express that in the form of functions. You also can't set a variable to something and then set it to something else later. So in purely functional languages, a function has no side-effects. The only thing a function can do is calculate something and return it as a result. If a function is called twice with the same parameters, it's guaranteed to return the same result. That's called referential transparency.

Data structures like lists, arrays, sequences, are called as collections. Functions like map and fold are Collection-oriented functions often have a lot of inherent parallelism If one can express computations with these primitives, then parallelism often becomes easy.

Haskell Functional Language
Haskell is a purely functional, non-strict language designed by an international group of researchers (Hudak et al. [1992]). The features of Haskell that stand out amongst the programming languages crowd are:

Haskell is a purely functional. Functions cannot have side effects or mutate data. For a given set of inputs (arguments) a function always gives the same result. The advantage of this model for reasoning about code is clear, but integrating input/output into the purely functional setting is a significant challenging task. An elegant solution was found in the form of monads. A monad is a structure that represents computations defined as sequences of steps. This allows the programmer to build pipelines that

- process data in a series of steps, in which each data goes through additional processing rules provided by the monad.
- Haskell is lazy. This refers to the evaluation method of the language. Most of languages use strict evaluation in which the arguments passed to a function are evaluated before the function is called, whereas in Haskell the arguments to a function are passed without evaluation, and only evaluated on demand. This behavior of Haskell also has benefits for reasoning about programs, but more than it serves as a barrier to prevent the leakage of impure non-functional features into the language, such features fundamentally cannot work in conjunction with lazy semantics.
- Haskell is statically typed. Which means, the compiler knows which piece of code is a number or string and so on? So lot of possible errors is caught at compile time. Haskell uses a very good type system that has type inference. It means that you don't have to explicitly label every piece of code with a type because the type system can intelligently figure out a lot about it. If you say a = 6 + 4, you don't have to tell Haskell that a is a number. Type inference also allows the code to be more general. If a function you make takes two parameters and adds them together and you don't have to explicitly state their type, the function will work on any two parameters that act like numbers.

Glasgow Haskell Compiler

The GHC started as part of an academic research project funded by the UK government at the beginning of the 1990s, with some goals in mind:

- To make freely available a robust and portable compiler for Haskell that generates high performance code.
- To provide a modular foundation, that other researchers can extend and develop.
- To learn how real programs behave, so that we can design and build better compilers.

High-Level Structur
At the highest level, GHC can be divided into three distinct chunks:

- The **compiler** itself, whose job is to convert Haskell source code into executable machine code.

  The **Boot Libraries**. GHC comes with a set of libraries called as the boot libraries because they constitute the libraries that the compiler itself depends on. With these libraries in the source tree means that GHC can bootstrap itself. Some of these libraries are very tightly coupled to GHC, because they implement low-level functionality such as the `Int` type in terms of primitives defined by the compiler and runtime system. Other libraries are more high-level and compiler-independent, such as the Data.Map library.

- The **Runtime System** (RTS). This is a large library of C code that handles all the tasks associated with running the compiled Haskell code. It including garbage collection, thread scheduling, exception handling and so on. The RTS is linked into every compiled Haskell program. The RTS is the most significant module of GHC, and the design decisions made here are responsible for some of Haskell's key strengths, such as its efficient support for concurrency and parallelism.

**CONCLUSIONS**

The modularity is the key to successful programming, so functional programming offers important advantages for software development as it supports the data parallelism. The pure functional Language like Haskell along with GHC compiler will match up with the multi core hardware.

**REFERENCES**

[1] Functional Programming and Parallel Computing
http://www.idt.mdh.se/kurser/DVA201/slides/parallel-4up.pdf

http://www.aosabook.org/en/intro2.html#marlow-simon

[3] Purely Functional Data Structures
Author: Chris Okasaki, Columbia University, New York


**Author[s] : Vandana Sunag**
KLE’s BCA, P C Jabin Science College, Vidyanagar Hubballi- 580021
Mail-Id: vsunag08@gmail.com
Mobile no : 9686876100

Sameeksha Irkal
KLE’s BCA P C Jabin Science College, Vidyanagar Hubballi- 580021
Mail-Id: irkalsamiksha@gmail.com
Mobile no : 7204305163