Functional programming and multicore architecture

Asked 16 years, 2 months ago Modified 14 years, 5 months ago

Viewed 6k times

18

I've read somewhere that functional programming is suitable to take advantage of multi-core trend in computing. I didn't really get the idea. Is it related to the lambda calculus and von neumann architecture?

f# functional-programming multicore

Share

Improve this question

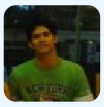
Follow

edited Jul 1, 2010 at 15:31

Chris Ballance

34.3k ● 26 ● 105 ● 152

asked Oct 8, 2008 at 3:59



lonegunman

1,005 ● 12 ● 14

9 Answers

Sorted by: Highest score (default)

13

Functional programming minimizes or eliminates side effects and thus is better suited to distributed programming. i.e. multicore processing.

In other words, lots of pieces of the puzzle can be solved independently on separate cores simultaneously without having to worry about one operation affecting another nearly as much as you would in other programming styles.

Share Improve this answer Follow

answered Oct 8, 2008 at 4:02

Chris Ballance
34.3k ● 26 ● 105 ● 152

@ja: I criticized some of their recent benchmark results here: flyingfrogblog.blogspot.com/2010/06/… – J D Jun 5, 2010 at 10:31

13

One of the hardest things about dealing with parallel processing is locking data structures to prevent corruption. If two threads were to mutate a data structure at once without having it locked perfectly, anything from invalid data to a deadlock could result.

In contrast, functional programming languages tend to emphasize immutable data. Any state is kept separate from the logic, and once a data structure is created it cannot be modified. The need for locking is greatly reduced.

Another benefit is that some processes that parallelize very easily, like iteration, are abstracted to functions. In C++, You might have a for loop that runs some data processing over each item in a list. But the compiler has no way of knowing if those operations may be safely run in parallel -- maybe the result of one depends on the one before it. When a function like `map()` or `reduce()` is used, the compiler can know that there is no dependency between calls. Multiple items can thus be processed at the same time.

Share Improve this answer Follow

answered Oct 8, 2008 at 4:05
John Millikin
201k ● 41 ● 215 ● 227

"But the compiler has no way of knowing if those operations may be safely run in parallel". Does this mean that *by default* the same loop will run faster in functional language like Scala or Haskell compared to an imperative language? – Aventinus Feb 15, 2016 at 14:48

@Aventinus theoretically, yes, but of course compilers of imperative languages can in some cases recognise that the body of the loop does not effect the computations in the loop, and then it will be able to parallelise. Also, the compiler of the functional language must be able to parallelise the calls (i.e., it has to be implemented). – user1544337 Jun 19, 2016 at 8:47

11

I've read somewhere that functional programming is suitable to take advantage of multi-core trend in computing... I didn't really get the idea. Is it related to the lambda calculus and von neumann architecture?

The argument behind the belief you quoted is that purely functional programming controls side effects which makes it much easier and safer to introduce parallelism and, therefore, that purely functional programming languages should be advantageous in the context of multicore computers.

Unfortunately, this belief was long since disproven for several reasons:

- The absolute performance of purely functional data structures is poor. So purely functional programming is a big initial step in the wrong direction in the context of performance (which is the sole purpose of parallel programming).
- Purely functional data structures scale badly because they stress shared resources including the allocator/GC and main memory bandwidth. So parallelized purely functional programs often obtain poor speedups as the number of cores increases.
- Purely functional programming renders performance unpredictable. So real purely functional programs often see performance *degradation* when parallelized because granularity is effectively random.

For example, the bastardized two-line quicksort often cited by the Haskell community typically runs thousands of times slower than a real in-place quicksort written in a more conventional language like F#. Moreover, although you can easily parallelize the elegant Haskell program, you are unlikely to see any performance improvement whatsoever because all of the unnecessary copying makes a single core saturate the entire main memory bandwidth of a multicore machine, rendering parallelism worthless. In fact, nobody has ever managed to write any kind of generic parallel sort in Haskell that is competitively performant. The state-of-the-art sorts provided by Haskell's standard library are typically hundreds of times slower than conventional alternatives.

However, the more common definition of functional programming as a style that emphasizes the use of first-class functions does actually turn out to be very useful in the context of multicore programming because this paradigm is ideal for factoring parallel programs. For example, see the new higher-order `Parallel.For` function from the `System.Threading.Tasks` namespace in .NET 4.

Share Improve this answer Follow

edited Jun 27, 2010 at 20:29

answered Jun 27, 2010 at 20:21

J D
48.6k ● 14 ● 174 ● 277

conventional language like F# ? – Surya Jul 1, 2010 at 15:33

Yes. Convention meaning "not purely functional" in that context. – J D Jul 1, 2010 at 21:07

8

When there are no side effects the order of evaluation does not matter. It is then possible to evaluate expressions in parallel.

Share Improve this answer Follow

edited Oct 8, 2008 at 6:09

answered Oct 8, 2008 at 4:06

Jonas
19.6k ● 10 ● 57 ● 67

6

The basic argument is that it is difficult to automatically parallelize languages like C/C++/etc because functions can set global variables. Consider two function calls:

```
a = foo(b, c);
d = bar(e, f);
```

Though foo and bar have no arguments in common and one does not depend on the return code of the other, they nonetheless might have dependencies because foo might set a global variable (or other side effect) which bar depends upon.

Functional languages guarantee that foo and bar are independant: there are no globals, and no side effects. Therefore foo and bar could be safely run on different cores, automatically, without programmer intervention.

answered Oct 8, 2008 at 4:03

DGentry

16.3k ● 8 ● 53 ● 66

4

All the answers above go to the key idea that "no shared mutable storage" is a key enabler to execute pieces of a program in parallel. It does not really solve the equally hard problem of finding things to execute in parallel. But the typical clearer expressions of functionality in functional languages do make it theoretically easier to extract parallelism from a sequential expression.

In practice, I think the "no shared mutable storage" property of languages based on garbage collection and copy-on-change semantics make them easier to add threads to. The best example is probably Erlang, that combines near-functional semantics with explicit threads.

answered Oct 12, 2008 at 19:18

jakobengblom2

5,843 ● 2 ● 28 ● 34

1

This is a little bit of a vague question. One perk of multi-core CPUs is that you can run a functional program and let it plug away serially without worrying about affecting any computing going on that has to do with other functions the machine is carrying out.

The difference between a multi-U server and a multi-core CPU in a server or PC is the speed savings you get by having it on the same BUS, allowing better and faster communication to the cores.

edit: I should probably qualify this post by saying that in most of the scripting I do, with or without multiple cores, I rarely see a problem in getting my data through hackish parallelizing, such as running multiple small scripts at once in my script so I'm not slowed down by things like waiting for URLs to load and what not.

double edit: Furthermore, a lot of functional programming languages have had forked parallel variants for decades. These better utilize parallel computation with some speed improvement, but they never really caught on.

Share Improve this answer Follow

answered Oct 8, 2008 at 4:04

Robert Elwell
6,668 ● 1 ● 30 ● 32

0

Omitting any technical/scientific terms the reason is because functional program doesn't share data. Data is copied and transfered among functions, thus there is no shared data in the application.

And shared data is what causes half the headaches with multithreading.

Share Improve this answer Follow

answered Oct 8, 2008 at 4:04

Robert Gould
69.7k ● 61 ● 191 ● 275

1 Functional programs almost always share data. – J D Jun 27, 2010 at 19:59

0

The book Programming Erlang: Software for a Concurrent World by Joe Armstrong (the creator of Erlang) talks quite a bit about using Erlang for multicore(/multiprocessor) systems. As the wikipedia article states:

> Creating and managing processes is trivial in Erlang, whereas threads are considered a complicated and error-prone topic in most languages. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks.

Share Improve this answer Follow

answered Oct 12, 2008 at 19:48

KeyserSoze
2,511 ● 1 ● 16 ● 17