Swift in HPC

Investigations in the use of Swift in High Performance Computing

These are not the Swifts you are looking for!

- http://swift-lang.org/
  - A simple tool for fast, easy scripting on big machines
- https://www.swift.com/
  - Swift code is standard format of Bank Identifier Codes
- https://www.boulder.swri.edu/~hal/swift.html
  - A solar system integration package
- http://taylorswift.com/
  - Developed in Pennsylvania – not sure of efficacy in HPC

What is Swift

- Swift has been years in the making
  - simplified memory management with Automatic Reference Counting (ARC)
    - tracks and manages your app's memory usage.
    - this means that memory management "just works" in Swift – you do not have to do the garbage collecting
    - https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html
  - feels familiar to Objective-C developers
  - friendly to new programmers
    - It supports playgrounds (think of emacs)

Introduction

- What's the point of this talk? Why am I here?
- What is HPC?
- Apple Swift programming language?
  - Why should we use it?
  - How can it be used now?
  - Is it ready for big time use in HPC
- Bibliography
- Questions and possible swift answers

Point of this talk

- Can we write programs in Apple swift on linux clusters (as we can in Fortran, C, C++, Java)?
  - We know we can use GCD (Grand Central Dispatch), LLVM, Clang, BLAS on Apple stuff
- Because Apple swift is open source can we write programs that are intrinsically parallel?

What is HPC

- High-performance computing (HPC) evolved to meet the demands for processing speed. HPC brings together several technologies such as computer architecture, algorithms, programs and electronics, and system software under a single canopy to solve advanced problems/programs effectively, reliably, and quickly. A highly efficient HPC system requires a high-bandwidth, low-latency network to connect multiple nodes and clusters. The term applies especially to systems that function above a teraflop.

Current state of HPC

- Architectures
- Models
- MPI and OpenMP libraries,
- Concurrency and threads

Software Crisis in HPC

- A supercomputer application and software are usually much more long-lived than a hardware
  - Hardware life typically four-five years at most.
  - Fortran and C are still the main programming models
- Complexity is rising dramatically
  - Challenges for applications on Petaflop systems
  - Improvement of existing codes will become complex and impossible - The use of O(100K) cores implies dramatic optimization effort
  - support of a hundred threads in one node implies new parallelization strategies

Software Crisis in HPC

- Programming is stuck
  - Arguably hasn't changed so much since the 70's still using Fortran, C
- Software is a major cost component of modern technologies.
  - The tradition in HPC system procurement is to assume that the software is free.
- There is the need for new community codes and languages – Apple Swift?

Architectures

- Vector Processors - Cray-1
- single instruction multiple data(SIMD), Array Processors
  - Goodyear MPP, MasPar 1 & 2, TMC CM-2
- Parallel Vector Processors (PVP)
  - Cray XMP, YMP, C90 NEC Earth Simulator, SX-6
- Massively Parallel Processors (MPP) - Cray T3D, T3E, TMC CM-5, Blue Gene/L
- Commodity Clusters
  – Beowulf-class PC/Linux clusters – Constellations
- Distributed Shared Memory (DSM) - SGI Origin
- HP Superdome
- Hybrid HPC Systems - Roadrunner
- Chinese Tianhe-1A system
- GPGPU systems

Programming Models

- Message Passing (MPI)
- Shared Memory (OpenMP)
- Partitioned Global Address Space Programming (PGAS) Languages
  - UPC, Coarray Fortran, Titanium
- Next Generation Programming Languages and Models
  - Chapel, X10, Fortress
- Languages and Paradigm for Hardware Accelerators
  - CUDA, OpenCL
- Hybrid: MPI + OpenMP + CUDA/OpenCL

Question to Matlab and Maple

- Are there any plans to produce SWIFT code from Matlab/maple?
  - We do not publicly comment on future product plans. You would need to ask Sales as they are the only people authorized to talk about such things.

Bindings

- Swift is “parallel” similar to what C and C++ are without special tools. You can use pthreads or dispatch (stable release in Swift 3.0).
- Currently there is no OpenMp or MPI bindings for Swift. But these should be coming soon.

From OpenMP

- The runtime can be built with gcc, icc or clang. However, note that a runtime built with clang cannot be guaranteed to work with OpenMP code compiled by the other compilers, since clang does not support a 128-bit float type, and cannot therefore generate the code used for reductions of that type
- The OpenMP runtime is known to work on ARM® architecture processors and PowerPC™ processors
- 32 and 64 bit X86 processors when compiled with clang, with the Intel compiler or with gcc, and also the Intel® Xeon Phi™ product family, when compiled with the Intel compiler.
- Ports to other architectures and operating systems are welcome.
- http://openmp.llvm.org/README.txt

Can I do this in Swift?

```
void simple(int n, float *a, float *b)
{
   int i;
/* simple OpenMP call */
#pragma omp parallel for
   for (i=1; i<n; i++) /* i is private by default */
      b[i] = (a[i] + a[i-1]) / 2.0;
}
```

- Not yet but you can do this:

Wrappers

- **Create a Swift Command Line Utility in Xcode**
  - First create a simple command line utility in Swift using Xcode.
- In Xcode go to File->New->Project
  - in the dialog that appears select Command Line Tool and click Next
- Choose a name for your project and type it in the Product Name field in the next dialog;
  - make sure Swift is selected in the Language field and lets call it *cli_swift* and click Next
- In the dialog that appears select a location for your project and click Create.
  - you will get a project consisting of one Swift source file, main.swift, which contains:

```
import Foundation
println("Hello, World!")
```

- comment out or remove the "import Foundation" line, since we are not using any of the Foundation features.
- click the Run button from the toolbar, or
  - selecting Run from the Product menu, or
  - by typing Command-R.
  - We see "Hello, World!" printed out in the output pane.

Wrappers

- **Create a simple C++ static Library Using GCC**
  - This could be done in Xcode..
  - The command for compiling/linking C++ code is g++. (or if you are using clang – clang++)
- Choose a location for the library and cd to that directory.
- Create the header file for the library, let's call it junk.h, with the following content:

```
class A{
public:
  A(int);
  int getInt();
private:
  int m_Int;
};
```

- Create the library's implementation file, call it junk.cpp, with the following content:

```
#include "junk.h”
A::A(int _i) : m_Int(_i) {}
int A::getInt() { return m_Int; }
```

- Compile junk.cpp to object file - junk.o, create a library - libjunkcpp.a, that contains the object file

```
$ g++ -c junk.cpp
$ ar r libjunkcpp.a junk.o
```

Wrappers – the “glue”

- **Add C++ Wrapper to the Xcode Project to provide the Swift-C++ Interface**
  - go back to Xcode and create a C++ file providing the "glue" between our C++ library and command line utility written in Swift.
  - The "glue" has to be in C++ because it will call C++ library code that was not written to be called from C.
  - The "glue" is, however, written in such a way that it can be called from C code, and things that can be called from C can typically be also called from Swift. See the comments in the code below.
- Add the static C++ library to the Xcode project. We need this in order to be able to call the library from the C++ wrapper ("glue").
  - Add the library's header file, junk.h, to our project by going to File->Add Files to "cli_swift"..., navigating to the file, and clicking Add.
- Go to File->New->File, select C++ file in the dialog that appears, and click Next.
- Type wrapper.cpp in the Name field, uncheck "Also create a header file," and click Next.
- In the dialog that appears choose a location for the new file and click Create.
- If at this point Xcode suggests that you create a bridging header, click Yes.
  - Otherwise create a bridging header manually by going to the "Swift Compiler - Code Generation" section under Build Settings and specify a header name on the "Objective-C bridging Header" line.

Wrappers – the “glue”

- Modify wrapper.cpp to have the following content:

```
#include "junk.h"
// extern "C" will cause the C++ compiler
// (remember, this is still C++ code!) to
// compile the function in such a way that
// it can be called from C
// (and Swift).
extern "C" int getIntFromCPP()
{
   // Create an instance of A, defined in
   // the library, and call getInt() on it:
   return A(314159).getInt();
}
```

Wrappers

- **Use C++ from Swift Code via the Wrapper**
- tell Swift about the getIntFromCPP() method.
  - Any C functions that we want to call from Swift must be declared in the bridging header, so we add the following to the bridging header:

int getIntFromCPP();

- Now we can call the method from Swift by adding the following to main.swift:

println(”The value of PI without a decimal is \(getIntFromCPP())")

- Run the project, and the output is:

$Hello, World!

$The value of PI without a decimal is 314159

Concurrency in Swift

- By default, when you make an application it runs the code in a single-thread environment, a main thread. For example, an iOS application would call the application: didFinishLaunchingWithOptions method on the main thread.
- A simpler example is an OS X Command Line Tool application. It has only one file: main.swift. When you start it, the system creates a single main thread and runs all the code in the main.swift file on that thread.
- For testing code, playgrounds are the best. By default, playgrounds stop after executing the last line of code and don't wait for the concurrent code to finish executing. We can change this behavior by telling the playgrounds to keep running indefinitely. To do that, include these two lines in the playground file:
- import XCPlayground
- XCPSetExecutionShouldContinueIndefinitely()

Threads

- A thread is the most low-level Application program interface (API).
- All the concurrency is built on top of threads and runs multiple threads.
- We can use *NSThread* from the Foundation framework.
  - The simplest way to do this is to create a new class with a method that will be the starting point for our new thread.

Threads (untangling)

- You can create a new thread in two ways,
  - Use *detachNewThreadSelector* function
  - or create an instance of *NSThread* and use the start function. We have to mark our run function with the @objc attribute because we use it as a selector when creating a thread, and NSThread is an Objective-C class that uses dynamic dispatch for method calling.

Solution for Threads

- Instead of solving our initial task that we wanted to run concurrently, now we are spending time managing the complexity of that concurrent execution system. Fortunately we don't need to do that as there is a solution: *Don't use threads*.
- But what can we use?
  - **GCD** (**Grand Central Dispatch**) is a high-level API that is built on top of threads
  - performs all aspects of thread management for you

Grand Central Dispatch (GCD)

- Queues
  - Main queue
    - let mainQueue = dispatch_get_main_queue()
  - Concurrent queue – concurrent and owns queues
    - func dispatch_get_global_queue(identifier: Int, flags: UInt) -> dispatch_queue_t!
  - Serial
    - let serialQ = dispatch_queue_create("my-s", DISPATCH_QUEUE_SERIAL)
- Tasks
  - Blocks of code
- Adding tasks to queues

Adding Tasks to Queues

- We have a queue and task that we want to run.
  - To run a task on a particular queue, dispatch it.
  - Two ways:
- **Synchronous**: dispatch_sync
  - dispatch_sync(queue: dispatch_queue_t, _ block: dispatch_block_t)
- **Asynchronous**: dispatch_async
  - dispatch_async(queue: dispatch_queue_t, _ block: dispatch_block_t)

Synchronous Dispatch

Synchronous dispatch submits a task for execution and waits until the task is done.

```
dispatch_sync(queue) { ... }
```

When you use a concurrent queue and dispatch a task to it synchronously, the queue can run many tasks at the same time, but the *dispatch_sync* method waits until the task you submitted is finished.

```
let queue = dispatch_get_global_queue(QOS_CLASS_BACKGROUND, 0)
dispatch_sync(queue) { print("Task 1") }
print("1 Done”)
dispatch_sync(queue) { print("Task 2") }
print("2 Done”)
```

More on Dispatch

- Never call the *dispatch_sync* function from a task that is executing in the same queue. This would cause a deadlock for the serial queue and should be avoided for concurrent queues as well.

```
dispatch_sync(queue) {
  dispatch_sync(queue) {
    print("Never called") // Don't do this
  }
}
```

Concurrency Guide

- GCD also has some powerful tools for synchronizing submitted tasks.
- Read the *Concurrency Programming Guide* article in the Apple library documentation at https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide

Timings

- There are, at least six ways to measure elapsed time in a Swift program.
  - NSDate
  - CFAbsoluteTime
  - NSProcessInfo.systemUptime
  - mach_absolute_time with mach_timebase_info
  - clock() in Posix Standard
  - times() in Posix Standard
- Or you can let Instruments do the heavy lifting

Timing Examples

NSDate completion block

```
public class func secElapsed(completion: () -> Void)
{
    let startDate: NSDate = NSDate()
    completion()
    let endDate: NSDate = NSDate()
    let timeInterval: Double =
endDate.timeIntervalSinceDate(startDate)
    println("seconds: \(timeInterval)")
}
```

Performance – How do we know we are doing well

- It has to be fast
  - Stopwatch – timing functions
  - metrics – Amdahl's law, Gustaftson-Barsis's law, Karp-Flatt metric, Isoefficiency relation
  - Write quality code – it has to be solid and flexible
  - Sweat the small stuff – arrays or sets?, string or int?
  - Don't optimize upfront
    - In the words of Donald Knuth: "*Premature optimization is the root of all evil*"

Some more Performance

- "Scotty we need more power"! (50th anniversary of the original Star Trek series)
  - What do we do when Scotty replies "I'm giving you all she's got"!
    - We can add more servers but this just delays for some time
- Or we can remove the issue that causes the performance problem.
  - For that, we need to identify the problem, the slow piece of the code, and improve it.
- The key metrics that impact an application's performance:
  - Operations' performance speed
  - Memory usage
  - Tertiary usage

Performance Issues

- Firstly don't optimize upfront, and secondly, measure first.
- Measure the code's performance characteristics and optimize only those parts that are slow.

Performance Analysis Formulas

- Amdahl's Law
  - Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \geq 1$. The maximum speedup Ψ achievable by a parallel computer with p processors performing the computation is:

$$\Psi \leq 1 / (f+(1-f)/p)$$

Performance Analysis Formulas

- Gustafson-Barsis's Law
  - Given a parallel program solving a problem of size *n* using *p* processors, let *s* denote the fraction of total execution time spent in serial code. The maximum speedup Ψ achievable by this program is:

$$\Psi \leq p + (1-p)s$$

  - Check out another possible speedup by Gustafson
    - The End of Error: Unum Computing - a new approach to computer arithmetic: the universal number (unum)

Performance Analysis Formulas

- Karp-Flatt Metric
  - Given a parallel computation exhibiting speedup Ψ on p processors, where $p>1$, the experimentally determined serial fraction e is defined to be:

    $e = (1/\Psi - 1/p)/(1 - 1/p)$

Performance Analysis Formulas

- Isoefficiency Relation
  - Suppose a parallel system exhibits efficiency $\varepsilon(n,p)$ where n problem size and p denotes the number of processors. Define $C = \varepsilon(n,p)/(1-\varepsilon(n,p))$. Let $T(n,1)$ denote the sequential execution time, and let $T_0(n,p)$ denote parallel overhead (total time spent by all processors performing communications and redundant computations). In order to maintain the same level of efficiency as the number of processors increases, problem size must be increased so that the following inequality is satisfied:

$$T(n,1) \geq CT_0(n,p)$$

Debugging

- Performance measuring in unit tests
  - When you create a new project, Xcode creates a unit test target for that project with the name ProjectName + Tests. You can read about testing in Xcode at: https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/testing_with_xcode.

Debugging –REPL

- **REPL** stands for **read-eval-print-loop**.
  - Swift REPL is an interactive Swift code interpreter that executes code immediately. To launch Swift REPL, open Terminal. app and execute this command:

    $ xcrun swift
- If there is an error the program will stop and you can examine and continue the code at that point.
- Writing code in the REPL console is not as convenient as in the modern Xcode IDE
- Apple has built more powerful tools,
  - such as Playgrounds, which has a nice source code editor and flexibility of Swift REPL.

LLDB

- LLDB is a high-performance command-line debugger. It is also available in Xcode. The easiest way to start it is to set a breakpoint and run the application. In the Xcode debug area view, you will find a console in which you can execute LLDB commands. To print the content of a variable, we can use the p LLDB command. Just run p with the variable name.
- LLDB is a very powerful debugger. You can read more about the LLDB debugger at https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide.

Conclusion

- You can do parallel computing with Swift (albeit in a limited way)
  - Use threads
  - Use wrappers
- Powerful debugger (LLDB) and editor in REPL
- Profiler – Instruments

Bibliography

- https://www.safaribooksonline.com/library/view/swift-high-performance/
- https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html#//apple_ref/doc/uid/TP40014097-CH3-ID0
- https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html
- https://developer.apple.com/videos/play/wwdc2016/720/ - concurrent programming in Swift3

Some words to leave with you

Et factum est, fieri cito

(Well done is Swiftly done)

Caesar Augustus