HONEY, I SHRUNK THE CLUSTER! PARALLEL COMPUTING AND MONTE CARLO SIMULATIONS ON iPOD TOUCHES, iPHONES AND iPADS*

Michael P. Rogers
Computer Science / Information Systems
Northwest Missouri State University
Maryville, MO 64468
Michael@nwmissouri.edu

ABSTRACT

One of the more intriguing new frameworks in the iPhone 3 Software Developers Kit (SDK) is GameKit, designed to support multiplayer gaming on the iPod Touch/iPhone/iPad family of devices. With surprisingly little effort, it is possible to adapt GameKit to set up a cluster of devices to perform parallel computing. This paper outlines what is required, and how this might be used in an intriguing Monte Carlo simulation.

1. INTRODUCTION:

There has been much written in recent years on the topic of cluster computing — harnessing multiple computers to a common purpose. While these clusters are typically built with discarded computers, it seems reasonable to investigate whether a useful cluster can be built using the iPod Touch/iPhone/iPad family of devices (abbreviated henceforth as the iFamily).

On the basis of CPU specifications alone, one might conclude that the outlook is bleak: while perfectly suited for their intended purpose, iFamily devices are slow compared to even the lowliest of computers that one might find in an inexpensive Beowulf cluster. For example, the iPhone 3GS CPU runs at 600 MHz. However, one of the hallmarks of cluster computing is that the individual computers need not be particularly powerful: it is the size and parallel nature of the cluster that gives it its speed.

---

* Copyright © 2010 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

2. CLUSTER REQUIREMENTS:

2.1 Hardware

The cluster will be based on iFamily devices running iPhone OS 3.0 or higher, connected via Bluetooth or Wi-Fi. It is pleasing to observe that, with Bluetooth, absolutely no additional hardware is required. Regardless of the networking technology, thanks to Bonjour, Apple's Zero-Configuration Network Device Discovery technology, assembling the cluster is merely a matter of placing the devices in proximity, and launching the apps.

A trivial cluster can be constructed with as few as 2 devices, but more is better. For embarrassingly parallel problems, it is possible to mix-and-match different devices (e.g., 2 iPhones and 3 iPod Touches). Note that the iPod Touches and iPhones must be at least second generation. Since this cluster will use a client-server architecture, it makes sense to make the server the fastest of the available iFamily devices.

2.2 Development Considerations

Developing iFamily apps, such as the software that will run on the cluster, has its own unique hardware requirements. Specifically, development must be done on Macintosh computers running Mac OS X 10.5 or higher. Thankfully, the requisite software — the iPhone 3 SDK, a free download from developer.apple.com — can run on even the least expensive Intel-based Macintoshes, so obtaining appropriate hardware should not be a huge fiscal obstacle.

For testing the server app, it will be necessary to run multiple client apps. This could be done on multiple Macintoshes, each running an iPhone Simulator; or, it could be done with 1 Macintosh, with multiple iFamily devices attached. Unfortunately, Apple does not officially support running multiple copies of the iPhone Simulator on one Macintosh.

3. A BRIEF OVERVIEW OF OBJECTIVE-C

This section is for those unfamiliar with the language of iFamily apps, Objective-C. In this obscure but easy-to-learn language, objects are sent messages by enclosing the object reference and message in [ ], and arguments are labeled. Consider an example:

```
[person setAge:25 firstName:@"Monty" isFamous:YES];
```

The object is person: the method follows. It has three arguments — an int (25), string (@"Monty"), and BOOL (YES). In Objective-C, string objects are preceded with an @, to distinguish them from ordinary C strings. Booleans have the value YES or NO. The name of the method is the concatenation of its labels. In this case, the method's name is setAge:firstName:isFamous, a bit lengthy but certainly easy to understand.

Instance methods are preceded by a –, and class methods by a +. Data types are enclosed in (). For instance, the setAge:firstName:isFamous method might look like this:

White space is ignored: the above could have been on one line, as:

```
-(void) setAge:(int) age
     firstName:(NSString *) fName
      isFamous:(BOOL *) famous {
// method body omitted
}
```

```
-(void) setAge:(int) age firstName:(NSString *) fName
isFamous:(BOOL *) famous
```

Objective-C divides object instantiation into two components— memory allocation and initialization. They are typically nested in one statement so that, for instance,

```
Person * person = [[Person alloc] initWithAge:25
birthPlace:@"Toronto"];
```

allocates a Person object, and then initializes two instance variables.

4. GAMEKIT

GameKit is a refreshingly compact framework, consisting of a mere 6 classes, which first appeared in iPhone OS 3.0. While designed to facilitate multiplayer games, it is more versatile: GameKit makes it possible to establish peer-to-peer and server-client connections in just a few lines of code.

At the heart of GameKit is a GKSession. GKSessions are instantiated in an app, discover other GKSessions running in apps on other iFamily devices on a local network, and then communicate with those GKSessions. The GameKit API calls any device on the network running a GKSession a peer.

GKSessions can establish themselves as **servers**, **clients**, or **peers**, and behave accordingly: servers advertise their services, and clients connect to them; peers both simultaneously advertise their services and connect to other peers. We use the term server to mean a GKSession running as a server; similarly for clients.

4.1 Servers

GKSessions that advertise as servers do so via a SessionID, a simple string. Clients wishing to connect to a particular server, use the SessionID to locate it.

Setting up a GKSession requires a mere 4 statements (found in the app's constructor):

The first statement instantiates a GKSession, with a Session ID of "Monty". It is stored in the instance variable session. [Notice the use of **self** rather than **this** as a keyword].

The second statement establishes a delegate. Delegation is a ubiquitous design pattern in iFamily apps that makes it unnecessary to extend a class and override its

```
self.session = [[GKSession alloc]initWithSessionID:@"Monty"
displayName:nil
sessionMode:GKSessionModeServer];
self.session.delegate = [[MyGKSessionDelegate alloc]init]; // for peer requests
[session setDataReceiveHandler:self.censusTaker withContext:nil]; // for data
self.session.available = YES; // now the server starts advertising on the network
```

methods to customize behavior. Instead, those methods are defined in a separate class, the delegate.

Specifically, in this case, the method to handle requests from clients to connect to the session, is defined not in a subclass of GKSession, but in the MyGKSessionDelegate class that the GKSession references. The method, which here accepts the request, is short:

```
// Defined in MyGkSessionDelegate
// Called when a client requests to connect to this server
session
- (void) session:(GKSession *)session
didReceiveConnectionRequestFromPeer:(NSString *)peerID{
NSError * error;
bool result = [session acceptConnectionFromPeer:peerID
error:&error];
// error handling code deleted
}
```

The third statement sets up the data handler — an object that defines a single method, `receiveData:fromPeer:inSession:context:`, which will be invoked when new data arrives from another peer. The censusTaker, an instance of the CensusTaker class, is responsible for tallying results from the clients: we will discuss it in section 5.

The last statement causes the session to start advertising itself by setting the available property. The session is now visible throughout the cluster. The actual exchange of information between client and server involves the `sendDataToAllPeers:withDataMode` and `receiveData:fromPeer:inSession:context:` methods which we will illustrate, in context, in section 5.

4.2 Clients

A GKSession acting as a client uses the same 4 statements as the server, with just one minor change: instead of GKSessionModeServer, it uses GKSessionModeClient. The difference is in the delegate. It must define a method, session:peer:didChangeState, which will be invoked when peers on the network have changed — most critically when a server session starts advertising. When it detects a server has appeared, it tries to connect to it — invoking the method session:didReceiveConnectionRequestFromPeer: defined above. Here is the code:

```
// A required method in the GKSessionDelegate protocol
// It is invoked when a peer changes state
- (void)session:(GKSession *)pSession
      peer:(NSString *)peerID
didChangeState:(GKPeerConnectionState)state{

      if(state == GKPeerStateAvailable){
            [pSession connectToPeer:peerID withTimeout:30];
      }
}
```

5. AN EXAMPLE: THE CLUSTER AS A SIMULATION TOOL IN THE CLASSROOM

For a demonstration, we have chosen to simulate the famous Monty Hall problem, in which contestants on a game show are presented with three doors. Behind one is a valuable prize, behind two are so-called zonk prizes. A contestant chooses a door, which remains unopened. Then Monty Hall opens a second door to show a zonk prize. The contestant is asked if they wish to stay with their current door, or switch. This simulation investigates the probability of success if they stay or if they switch.

The code for the actual simulation is straightforward, except the if statement: it is correct, and doubtful readers may wish to consult the appropriate reference in section 7.

```
- (NSString *)performSimulation{
        double numWinsStaying = 0;
        double numWinsSwitching = 0;
        int winningDoor, chosenDoor;

        for(int i=0; i < numGames; i++){
                winningDoor = [self randBetween:1 and:3];
                chosenDoor = [self randBetween:1 and:3];
                if(chosenDoor == winningDoor)
                        numWinsStaying++;
                else
                        numWinsSwitching++;
        }
        return [NSString stringWithFormat:@" %f / %f", // results delimited by a /
        numWinsStaying/numGames * 100, numWinsSwitching/numGames *
        100];
}
```

As noted previously, client and server communication involves repeated invocations of `sendDataToAllPeers:withDataMode` and `receiveData:fromPeer:inSession:context:`

The server begins, telling all the clients how many games to play:

```
NSData * numGamesToSimulate = [self . numGamesTF .text //
get # games as text then
           dataUsingEncoding :NSASCIIStringEncoding ]; //
           convert it
[ session  sendDataToAllPeers :numGamesToSimulate // send it
       withDataMode : GKSendDataReliable
           error :&error];
```

In the first line, the number of games is extracted from a text field (numGamesTF) as text, and then encoded into a generic container known as NSData. That NSData object is then broadcast to all the clients on the network.

Each client has a data handler, and the broadcast invokes its method:

```
- ( void) receiveData:( NSData  *)data
             fromPeer:( NSString  *)peer
            inSession: ( GKSession  *)session
              context:( void  *)context{
     NSString  * gamesStr = [[ NSString  alloc ]
                initWithData :data // NSData -> string
                encoding : NSASCIIStringEncoding ];
     numGames  = [gamesStr  intValue ]; // string -> int
     NSString  * resultsToSend = [ censusTaker
     generateResults ]; // do computation
     NSData  * dataToTransmitBackToServer = [resultsToSend
                // string -> NSData  dataUsingEncoding :
                NSASCIIStringEncoding ];
     [ session  sendDataToAllPeers
                :dataToTransmitBackToServer // send it
                withDataMode : GKSendDataReliable error
                :&error];
}
```

As each client in turn reports back, the text — delimited by a '/' — is parsed and the results added to those already gathered. [This code is essential, but not particularly instructive, and so not included here.] The display appears on the server, but of course, if desired, the results can then be echoed back to the clients. The cost of all the extra networking traffic has a somewhat detrimental affect on performance.

Although the Monty Hall problem is embarrassingly parallel, the time to solve it is not reduced to 1/n on an n-peer network, but it is of the order of 1/n. And the ease with which the network can be constructed, to say nothing of its novelty, makes the iFamily

cluster worthy of exploration in a variety of Computer Science / Computational Science classes.

6. CONCLUSION

While not the most powerful cluster, the iFamily cluster has several compelling features. First, students can take pride in helping assemble the cluster (even though the assembly consists of merely showing up with an iFamily device, loading, and launching an app: those students capable of writing the app will, of course, be able to take much more pride). Secondly, if the cluster is used for simulation in the classroom, the students can observe intermediate results, in the palm of their hands, as they appear on their iFamily device. This immediacy is enticing. Finally, because the cluster is so easy to use, and can be applied in so many situations, it is easy to envision a situation where two classes collaborate: a CS class writes the cluster software for use in another class, in another discipline.

GameKit remains a work in progress, and both it, and the documentation, are in a state of flux. In addition, for those planning to incorporate this into a course, it may take a considerable amount of ramp-up time to learn Objective-C and the associated frameworks. However, many of your students will have iFamily devices, and may be eager to program them, and they may be spurred by the end result — building a large cluster can be done more readily with this technology than any other.

7. REFERENCES:

Apple, Inc. *Game Kit Framework Reference*. Apple Inc., 2009.

Cochran, Stephen G. *Programming in Objective-C 2.0, 2nd Edition*. Addison-Wesley, 2008.

Gardner, Martin. *Aha! Gotcha: Paradoxes to Puzzle and Delight*. W.H. Freeman & Co., 1982.

IEEE. IEEE Task Force on Cluster Computing. www.ieeetfcc.org, retrieved March 22, 2010.

Mark, Dave and Jeff LaMarche. *More iPhone 3 Development*. Apress, 2010.

Prins, Philip R. Teaching Parallel Computing Using Beowulf Clusters: A Laboratory Approach. *Journal of Computing Sciences in Colleges*, (20), 55-61, 2004.