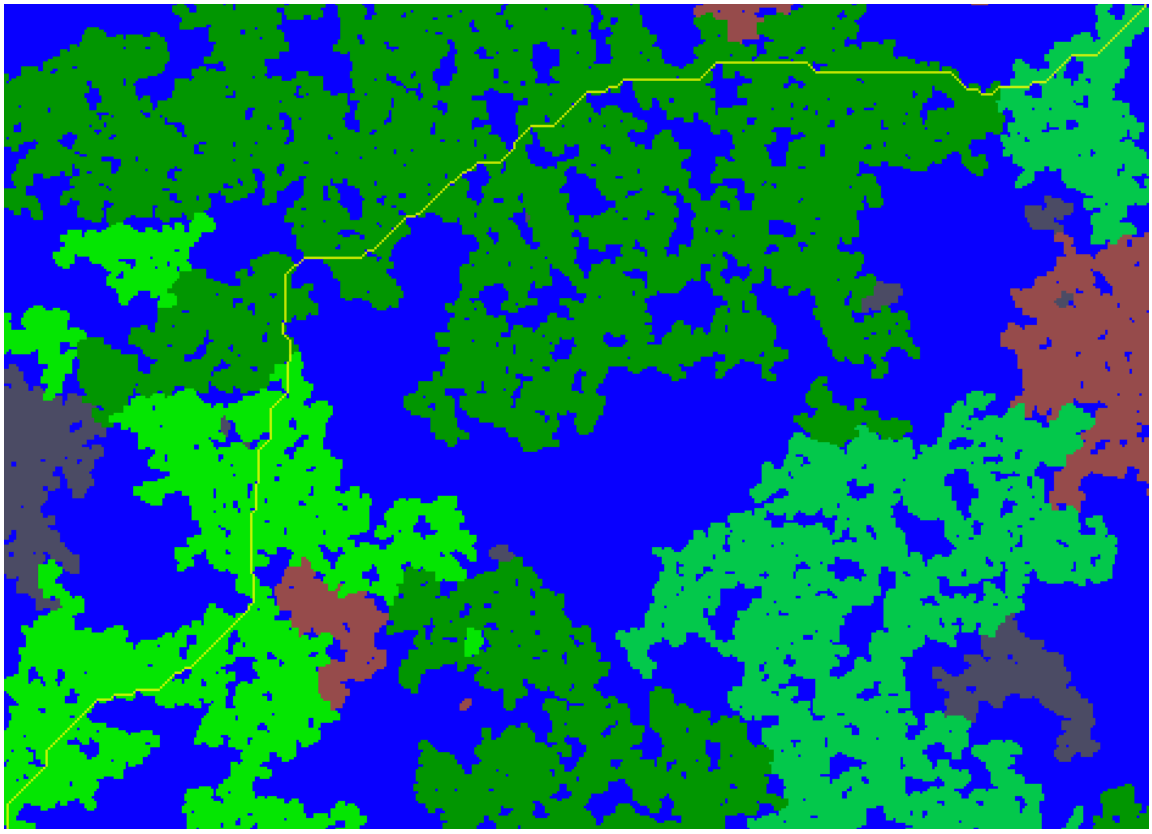
Dijkstra's Algorithm, in Parallel

An examination of using multicore processors and OpenCL for path finding.

Chris Steele
DePaul University, Chicago IL • March 17, 2012



Dijkstra's Algorithm

What is it and how does it work?

PART ONE

Dijkstra's algorithm is a well known, serial, algorithm to find the shortest distance between two nodes in a directed graph where the cost of moving from one node to another is a non-negative value.

In Plain English

What Dijkstra's algorithm does can be explained easily using a real world example. Imagine you want to get from one point in the city to another by using the shortest route possible. If you take a map and find every place two roads intersect and call those nodes, you'll have all the nodes of the graph Dijkstra's will work on. The cost of going from one node to another will be the distance of the road from one intersection (or node) to the other. Note here that we're using distance for the cost, however this could be anything. If we're looking for the fastest route instead of the shortest, then we could take the distance divided by the speed limit for the cost from one node to the next. And if traffic information were available, we could even factor then in by using the average current speed limit. Whatever the case, there will always be some cost associated from going from one intersection to the next.

What Dijkstra's algorithm does is start at the intersection you are currently at. It then looks at all intersections you can get to directly (that is, without traveling through a another intersection). The intersection that is the closest is favored over the others. At that point we now consider all intersections we can get to directly from this new intersection as well as the ones from the older intersection. We consider all of them and pick the one with the shortest cost (or distance). We then consider from this new intersection. We continue this process until we find our target intersection.

The Pseudo Code (or not-at-all plain english)

The following is Pseudo Code for Dijkstra's algorithm [1].

```
01: shortest_paths( Graph g, Node s )
02:    initialize_single_source( g, s )
03:    S := { 0 }        /* Make S empty */
04:    Q := Vertices( g ) /* Put the vertices in a PQ */
05:    while not Empty(Q)
06:        u := ExtractCheapest( Q );
07:        AddNode( S, u ); /* Add u to S */
08:        for each vertex v in Adjacent( u )
09:            relax( u, v, w )

10: relax( Node u, Node v, double w[][] )
11:    if d[v] > d[u] + w[u,v] then
12:       d[v] := d[u] + w[u,v]
13:       pi[v] := u
```

Using our example from above, we'll walk through the Pseudo code. Line 2 simply sets up all the data structures used in the algorithm. One important thing it does is set every entry in 'd' to infinity and every entry in 'pi' to empty. These will be discussed shortly. Line 3 initializes S to be empty, this will eventually include all examined nodes as the algorithm progresses. Line 4 adds all the unexplored vertices to the set Q. This is one variation of the algorithm, we could add this as we visit them as well, the outcome is the same. Line 5 simply states that we're going to examine all the vertices (or nodes or intersections in our example above) in the graph. When we've checked them all, we're done. Line 6 gets the closest node (the closest intersection in the example above). Line 7 adds that node to our solved set since we have the shortest path to it from the source. Line 8 and 9 then take all the nodes connected to this new node and "relaxes" or adds them to the nodes to be examined. It does this with the relax method defined on lines 10-13. The relax method works like this. If the distance to the new node is greater than the distance to the current node plus the distance from the current node to the new node, then we update it's distance to that value. Remember that distance to all nodes start out at infinity so the first time a new node is examined, it's distance is always updated to the distance to the current node plus the distance from the current node to the new node. In future iteration this distance is only updated if we find a shorter path. Lastly we update 'pi' to of the new node to be the current node. Pi is an array that simply tracks the node with the shortest distance that connects to our new node.

---

[1] Pseudo code taken from: http://www.cs.auckland.ac.nz/~jmor159/PLDS210/dijkstra.html

Once all nodes have been processed or we add our end node to S (not represented in this version of the pseudo code, but a simple optimization on the algorithm) then we're done. To find the complete path, we simply take the final node and check what node 'pi' points at for that node. We then look at that new node and look it up in 'pi' and so on. We walk the path backwards until we get to the starting node. We've found our shortest path.

Observations

We can make a few observations of this algorithm. First, the basic version of it examines all nodes in the graph. This isn't always optimal, so we almost always add a stop clause to end processing once we've found our target node and added it to S.

Second, and a bit more subtle, is that the running time of this algorithm is almost completely dependent on extracting the shortest distance from Q. In the most naive approach we'd have to search all elements of Q every time we needed the node with the shortest distance. This makes the running time of the extraction $O(N^2)$. Turns out, we don't want to do it that way. Most often we'll use a priority heap (in this case min heap) to reduce the cost of finding the minimum to $O(1)$. However we shift the cost to creating that priority heap which is $O(\log N)$, still much better than $O(N^2)$. With this optimization the entire algorithm runs in $O(E + N \log N)$ were E is the number of edges and N the number of nodes.

Lastly, and an important observation for my specific problem I'll attempt to solve, if we cap the distance we allow a node to cost then we can get a graph of all nodes that can be traveled to within that distance. In our example above we could find all the places we could go before running out of gas.

The Problem

Finding a path through a terrain map.

PART TWO

Dijkstra's algorithm is meant to solve a generic problem, get from one node to another for the least cost in a directed graph. That's what I'll be demonstrating here, but my graph has a very specific topology. I have two goals here. First is to find the shortest path from point A to point B. The second is to present valid points on a map that you could move to given a certain budget for that movement. In other words, all the points you can get to from A costing less than X.

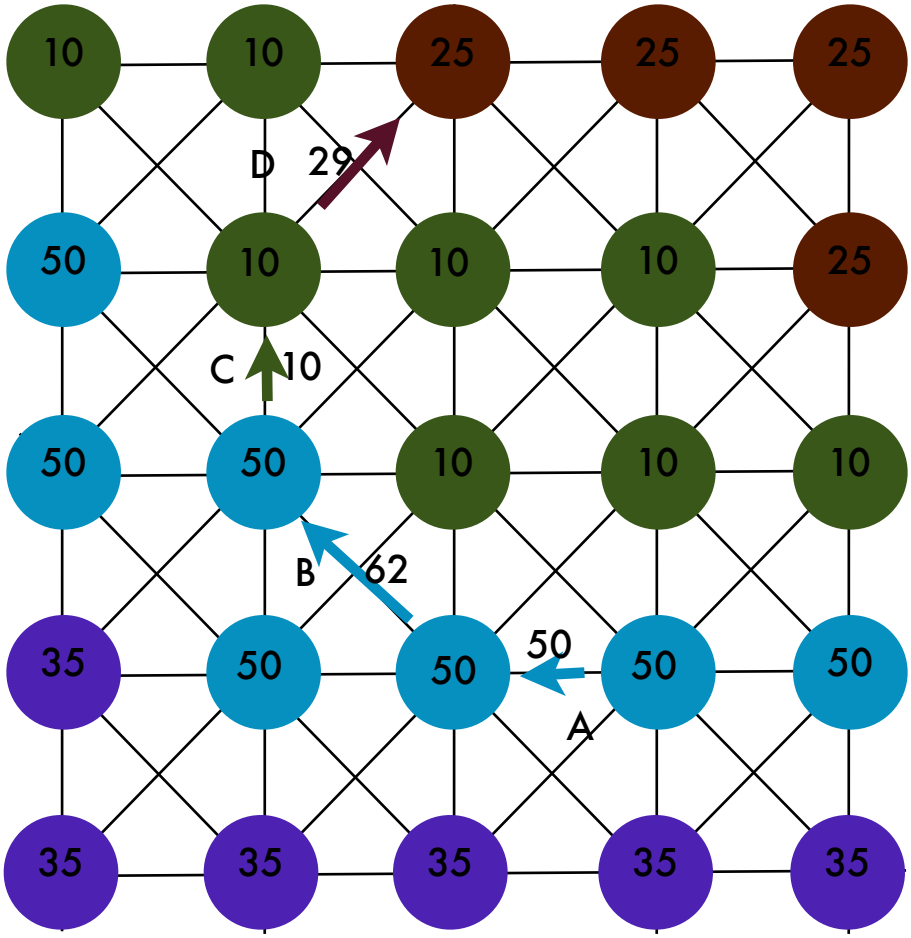
The Terrain Map

The graph I'll be using represents a terrain map. This map will be randomly generated but it follows very specific rules for it's layout and how it's connected.

The map is made up of a number of nodes or vertices laid out in a grid. Every node (except the edge nodes) have a node above, below, to the left and to the right of it. This also means that every node has four diagonal nodes as well. Diagonal movement can be toggled on and off in the simulation achieving slightly different end results. Each node connection is a two way connection, but in practice you'll never double back on a node you came from.

The cost to move from one node to the next is based on the terrain type of the node you are moving on to. If you are moving directly to an adjacent node, the cost is the straight cost of the terrain type. If you are moving diagonally the cost is equal to the cost of the node you are moving on to plus 41% of the average of the cost of the nodes you are "passing through".

The following example demonstrates the layout of the grid and the cost calculation.

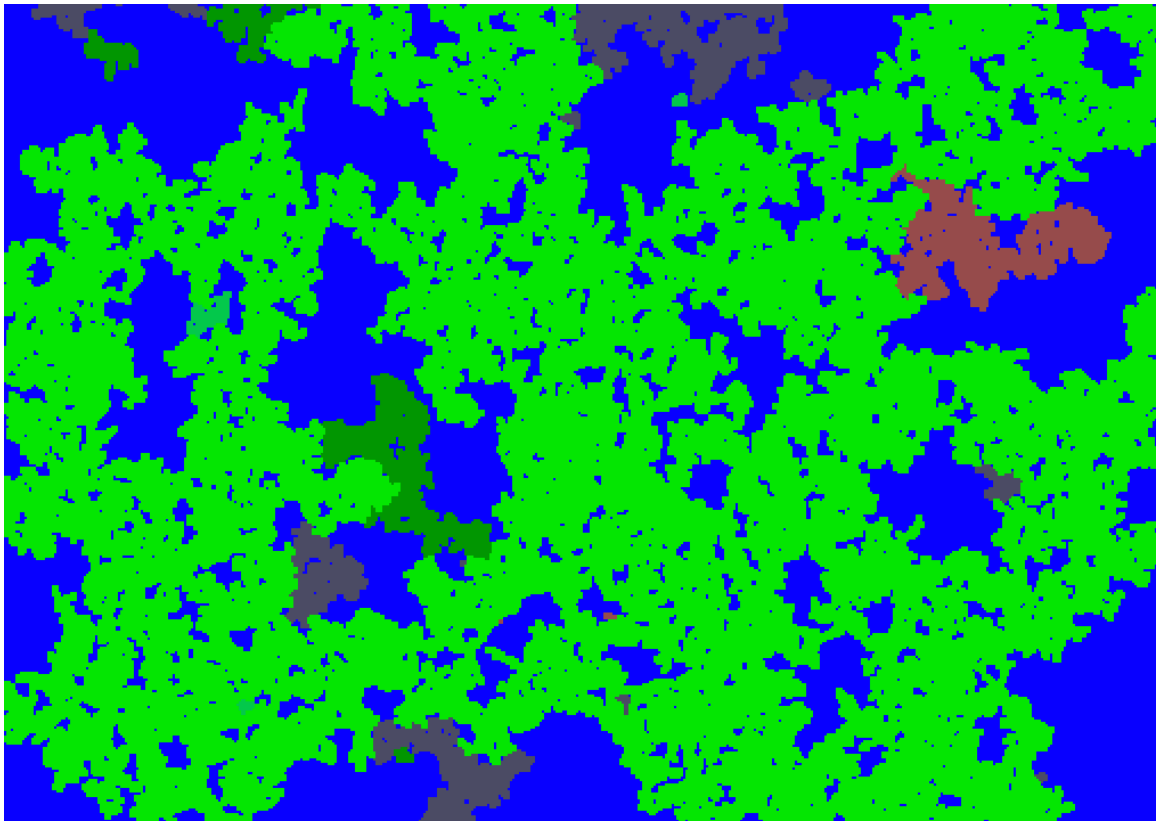


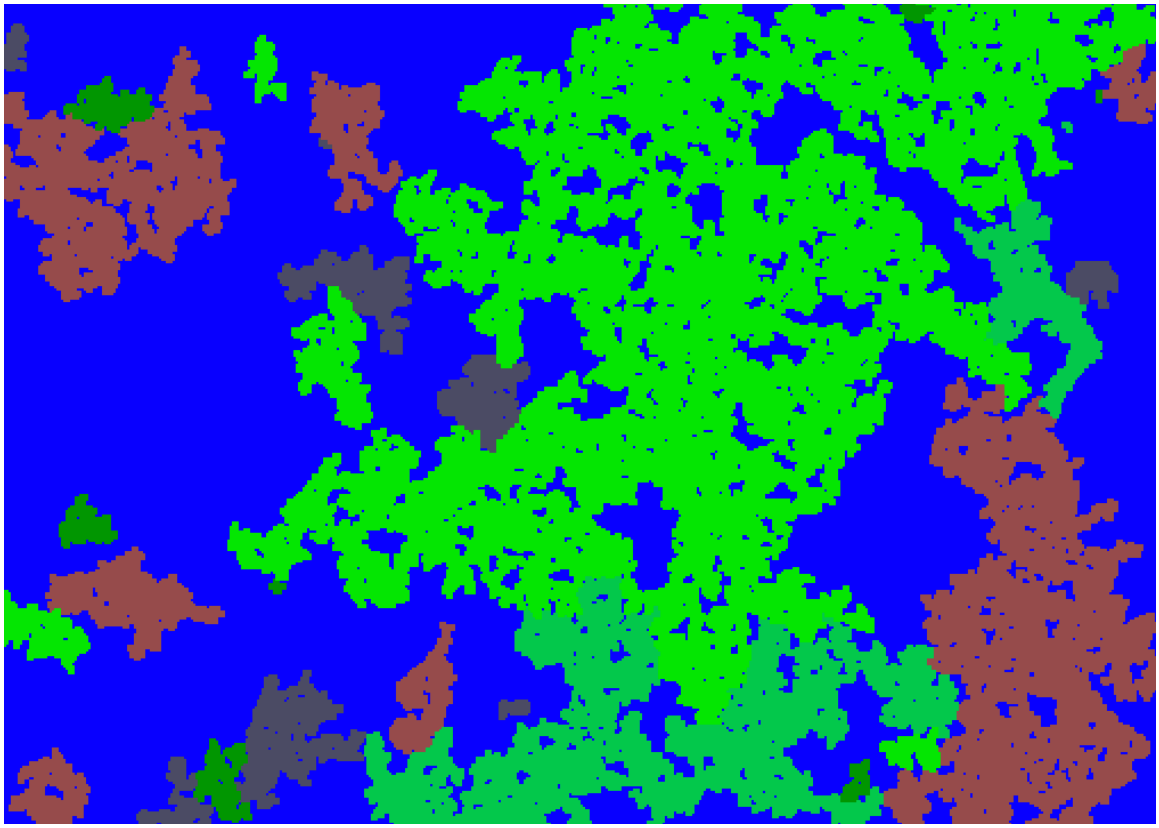
In the above example, green represents normal ground, the easiest terrain to move on. Red are rocks or mountains, fairly difficult to move on. Blue is water, very difficult to move on (we're assuming a man on foot). And purple is swamp, still very difficult to move on but not as bad as water.

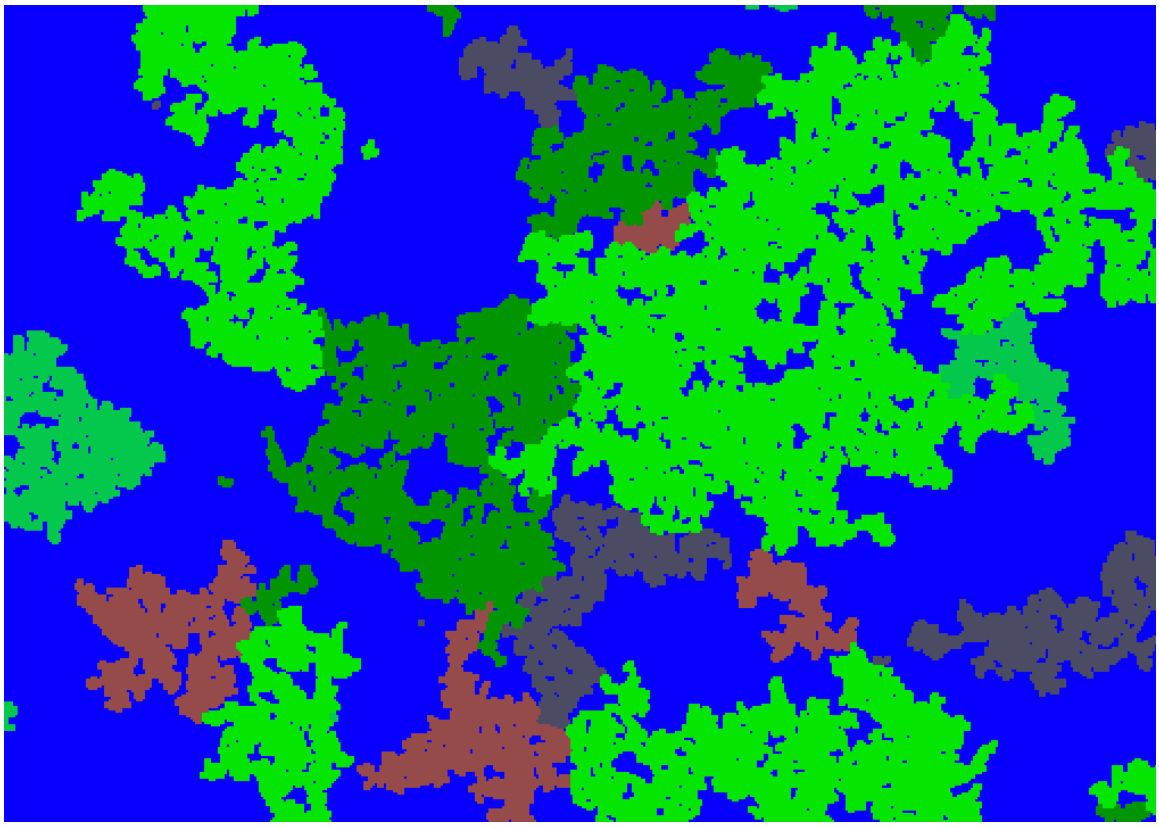
Although the path shown is not the shortest path between the start and end points, it does represent how the movement costs are calculated. Step A cost a straight 50 because it's adjacent movement from one node to the next, so the cost is the cost of the node we're moving to. Step B cost 62 because we must pay the cost of the node we're moving to, 50, plus 41% of the cost of the average of the two nodes we're moving through, 50 + 10 = 60/2 = 30 * .41 = 12. Step C is very cheap since it's direct movement on to ground. And step D is only slightly more expensive since ground is cheap and the diagonal is only four extra: 10+10 = 20/2 = 10 * .41 = 4.

Map Generation

Since this isn't an exercise in map generation algorithm (although that would be a very interesting parallel algorithm to examine) the generation of the map is pretty simple. Basically we start with a grid of water. We then "seed" random points with different terrain types. We then grow the seed by allowing the terrain type to spread with a 40% chance in any given direction. This leads to fairly interesting terrain generation for path finding. Although it doesn't make the most interesting maps to look at. Each map is defined by it's size and a seed so exact maps can be regenerated as often as need with only two numbers (size and seed). Here are some sample maps generated with this algorithm. All of these maps have a size factor of 2 meaning they measure 1024/2 x 768/2 (512 x 384). The test program was designed to run at 1024 x 768, or the resolution of the iPad. With the iPad 3 a size factor of 0.5 is needed to fill the screen. Size factors of 16 or 32 were used to test and debug the algorithms.







Applying Dijkstra in Serial

Our control case.

PART THREE

In order to measure any increase in speed gained by parallelizing the algorithm, we must first have a control timing based on the original algorithm.

First Implementation

As noted above in the description of Dijkstra's algorithm, the performance of the algorithm is directly related to how fast we can get the minimum distance node from the set of all nodes. My first attempted used the naive linear search of the entire set. As expect, with smaller maps the performance was adequate. but as the bigger maps were tested, the performance was changing logarithmically to the point were it took over 5 minutes to finish on the biggest map.

Second, Better, Implementation

Once the basic algorithm was working I did an optimization pass. The most major of the optimizations was added a priority heap for retrieve the minimum distance nodes. This took the running time of the largest map down from 5 minutes to 4 seconds. At this point the standard algorithm is so fast, that it'll likely be very hard to speed up.

Dijkstra using Blocks

Using Mac OSX and iOS Blocks for parallelization

PART FOUR

In Mac OSX 10.7 Apple introduced GCD (Grand Central Dispatch) and Blocks into the objective -C language. These sound perfect for parallel algorithms. In this section I take a pass at using blocks to implement Dijkstra's algorithm in parallel.

What Are Blocks?

The idea is simple, the application developer should not have to manage their own threads. What blocks allows the programmer to do is provide chunks of code (much like anonymous functions or closures) that can be run in parallel (if set to) and managed by the operating system via GCD. GCD will schedule the blocks on separate cores if available. Blocks have many modes and uses outside parallelization as well. A block can be passed as a handler to a method that is triggered at some point in the future. Or it can be executed serially in response to some event. The way they are used here are in concurrent mode or asynchronous mode. A few key things to note is that blocks can be scheduled in any order and may finish in any order. Of course there are ways to wait for a group of blocks to finish (via semaphores) but for the most part you just have a bunch of parallel functions all doing their own work. This is incredibly similar to OpenCL except were OpenCL runs on the GPU cores, blocks run on the CPU cores. At the very least they are an easy way to test out different parallel implementations of our algorithm.

The Inner Loop With Blocks

Dijkstra's algorithm does not lend itself to being implemented in parallel all that easily. There are two possible points in the algorithm that may be parallelized. The easiest is running the relax function (the inner loop) in parallel. So this is where I started.

The setup is very easy. We simply create a concurrent queue. The code in obj-C that does that is:

```
queue = dispatch_queue_create("Direction Concurrent Queue",
DISPATCH_QUEUE_CONCURRENT);
```

Note the string is a user defined queue name that shows up in the debugger to identify our queue. I've called it a Direction Concurrent Queue because, as you'll see next, the block runs through all eight edges, one for each direction. The block and the code that adds it to the queue is here:

```
    dispatch_apply(8, queue, ^(size_t d)
    {
        int xOff = u->x+xOffsetForDirection(d);
        int yOff = u->y+yOffsetForDirection(d);
        if(xOff<0 || yOff<0 || xOff>=WIDTH || yOff>=HEIGHT)
            return; // off the grid, ignore this edge

        DPFVert *v = &map[xOff][yOff];
        int vIndex = v->index;

        CGFloat distToV = terrainMovementPoints(v->terrain);
        if(u->x!=v->x && u->y!=v->y)
            distToV *= 1.4;

        if(dist[vIndex] > dist[uIndex] + distToV)
        {
            dist[vIndex] = dist[uIndex] + distToV;
            prev[vIndex] = u;
            [self insertVertP1:v];
        }
    });
```

Note that this is a single, C, function call. The first parameter is the number of times we will run this block. The next is the queue you'll run it on (our concurrent queue) and lastly the actual block (or function) that's being run. the function has a single parameter, d, that is the index of this block (0-7 because we have 8 total blocks being run). We can use d to figure out what direction we're heading and calculate our distance. Each block then updates its distance if necessary and then adds the node to the priority queue if needed. This last step requires some special care, since only one block can insert at a time. This is handled with a semaphore lock on the insert method.

The Results

Blocks are limited by how many may run at once. That limit is generally based on the number of cores, but GCD can also reduce the number if it has other, more important, processes that need to use the other cores. And in reality the relax method, as implemented above, is not slow. So the results were not surprising, but they were disappointing. In general I saw a 50% increase in processing time for the algorithm. The end result of the algorithm was identical to the serial

version, it was just slower. This can be attributed to a number of things. First, GCD is still threaded. And any threads have overhead, even if the kernel is managing them for you. Since this method is already so fast, the thread management over shadows the speedup we get from performing this method in parallel. Second, the slow part, inserting into the priority heap, is locked to one block at a time. So even if we had zero thread overhead, each block would still have to wait its turn to insert into the heap. Needless to say, our first pass implementation with blocks was a bust.

Dijkstra's using OpenCL

Using OpenCL and a GPU for parallelization

PART FIVE

Originally I wanted this project to be an OpenCL implementation for pathfinding on the iPad. Unfortunately, as of yet, Apple has not released a public OpenCL API for any iOS device. So instead the project was moved to the Mac platform where OpenCL does exist.

What Is OpenCL?

OpenCL is an open standard Apple pioneered that is meant to be a common language for running kernels (functions, programs, whatever you'd like to call them) directly on a GPU. OpenCL is based on a very strict implementation of C99. The idea is straight forward, even if the execution (at least in C) isn't. You create kernels that can then be dispatched to the hundreds or thousands of cores on a GPU that all run in parallel to solve your problem. So that's what I'll attempt to do in this section.

The Inner Loop With OpenCL

Using OpenCL poses a new challenge than uses blocks. With blocks I can use a simple semaphore to make sure the insert method into the min heap is handled one at a time. With OpenCL this isn't as easily done, nor would you want to. There are a few things we know about the problem space however that will help. No node will have more than eight edges. I'll use this knowledge to queue up inserts into the min heap. Each time a kernel needs to do an insert (at most one per kernel) it'll add it to the min heap array at an index equal to the current length plus its kernel id. It will *not* increment the count of the min heap. This is because reading and writing to the same global value at undeterminable times would cause random results. Instead we'll add a special loop after the relax loop that will check all 8 entries after the current length, and if it finds a valid index (not -1) it'll insert that element into the heap. This extra step is equivalent to the in time to the semaphore block above but allows us to make use of the OpenCL kernels as well. The following code segment represents the OpenCL kernel's version of relax:

```
"if(dist[vIndex] > dist[uIndex] + distToV)  \n" \
"{                                         \n" \
"    dist[vIndex] = dist[uIndex] + distToV; \n" \
"    prev[vIndex] = uIndex;                \n" \
"    heap[heapSize+d] = vIndex;            \n" \
"}                                         \n" \
```

Here 'd' represents the kernel id number ranging from 0 to 7. The heapSize is unchanged and the new index is simple placed at a fixed point in the array. After all the kernels complete we run this segment of code to truly add the indexes to the heap:

```
int checkMax = unsettledVertCount+DIRECTIONS;
for(int i = unsettledVertCount;i<checkMax;i++)
{
    if(unsettledVertsHeap[i]!=-1)
        [self insertVert:allVerts[unsettledVertsHeap[i]]];
}
```

Anytime we find a valid index within 8 (or whatever DIRECTIONS is defined as) we call the insertVert method. This keeps the heap up to date after each iteration.

The Results With OpenCL

For the same reason the block method above did not result in a speedup, the OpenCL method also did not result in a speedup. In fact, the extra overhead involved in copying the memory buffers to the shared reason made the results even slower (much much slower). If we observe that we can only run as many processes in parallel as we have edges connected to each node, then it's easy to see our problem. Even though OpenCL can offer us hundreds of cores to run on simultaneously, we're only ever using eight. Add to that the amount of setup time and how often we have to copy the buffers in and out of shared address space, and it's easy to see that a pure Dijkstra's adaptation using OpenCL isn't going to be optimal.

Adapting Dijkstra's Algorithm

A new algorithm inspired by Dijkstra that better fits the problem.

PART SIX

As mentioned above, Dijkstra's algorithm is not well suited for being implemented in parallel. What I will attempt to do in this section is adapt the ideas of Dijkstra's algorithm to better fit my problem and in the process create an algorithm that is better suited to be run in parallel.

The New Problem

Finding a path from point A to point B is useful if you know you want to go to point B. But what if you want to know all the points you can go to given a certain maximum number of movement points, X? Dijkstra's gets us very close to the solution of this problem. In fact, if point B is far enough away (great than X away) it solves this problem for us as a side effect to finding the shortest path. Using this knowledge I'm going make this side effect the main effect and change the algorithm from being one of path finding to one of range finding.

The New Solution

The idea is straight forward. You have X movement points. Each node you travel to costs some of those movement points. Starting at your current position, find all nodes you could move to given your current number of movement points. As a side effect, you'll have the shortest distance to each of the nodes in the solution set, so you'll still be able to get the shortest path from this version of the algorithm.

This modification has been contrived from the beginning for use with OpenCL and hundreds of cores working in parallel. It is a brute force algorithm where we have force to spare. The algorithm run in a number of phases where each phase all nodes examine the distance to its neighbor nodes and updates its own distance if one of its neighbors has reduced its distance to something shorter than before. Once a phase completes without any node updating its distance, the algorithm is complete.

Each node will track the shortest distance to itself (each, but the source, starting at infinity) and its terrain type. To optimize each node's kernel, we'll also track a flag that tells us if a neighbor has been updated since the last time this node's kernel ran. If that flag is not set, the kernel exits with out doing any further processing. If it is set, then it examines all it's neighbors, finds the minimum, adds its own cost (based on its terrain type), and updates its distance to this new value. Finally, it tells all its neighbors to update themselves at their first opportunity (this could be in the same phase depending on the order the kernels run or the next phase). And lastly it'll set a global completion flag to false.

Since all these processes are happening in parallel it's important to not to have any variables that require the value read from it to write back to something else. When a node updates it's distance from it's neighbors, it simply uses the current value in that neighbor. When a node updates itself, it'll then notify its neighbors so that anytime after that notification the updated value is available. The global flag can only be set to false (i.e. not done) and is never read by the nodes. It is read and reset by the master processes. Because of this we can take advantage of as many cores as are available to us. And by only having a single data structure we can reduce the amount of memory that gets copied between processing units.

The New Implementation

This is the kernel code for my version of the algorithm.

```
__kernel void altDijkstraWork(
                              __global DPFNode* Grid,
                              __global int*     widthB,
                              __global int*     heightB,
                              __global int*     done
                              )
{
    int k = get_global_id(0);
    int width = *widthB;
    int height = *heightB;

    if(Grid[k].shouldProcess)
    {
        *done = 0;

        int notifyNeighbors = 0;
        for(int x=-1;x<2;x++)
            for(int y=-1;y<2;y++)
                if(x || y)
                {
                    int index = k+y*width+x;
                    if(index>0 && index<width*height)
                    {
```

```
                        float newDistance =
Grid[index].distanceFromSource;
                        if(x!=0 && y!=0) // diagonal movement
                            newDistance +=
(terrainMovementPoints(Grid[k].terrain)*1.4);
                        else
                            newDistance +=
terrainMovementPoints(Grid[k].terrain);
                        if(newDistance < Grid[k].distanceFromSource)
                        {
                            Grid[k].distanceFromSource = newDistance;
                            Grid[k].nearestNeighborIndex = index;
                            notifyNeighbors = 1;
                        }
                    }
                }

        if(notifyNeighbors)
            for(int x=-1;x<2;x++)
                for(int y=-1;y<2;y++)
                    if(x || y)
                    {
                        int index = k+y*width+x;
                        if(index>0 && index<width*height)
                            Grid[index].shouldProcess = 1;
                    }

        Grid[k].shouldProcess = 0;
    }
}
```

The Grid and done parameters need to be global since each kernel may change those values. Width and height do not need to be global, but I ran into some runtime errors when they weren't so they were changed over to solve the problem.

The first thing each kernel does is check to see if its node needs to be processed at all. If not, it simply exits. To start off with, all nodes are marked as should be processed and the source node gets a distanceFromSource value of 0. From there, the kernel takes over and the algorithm "settles" into a solution.

There are two sections to the main body of the kernel. The first looks at all it's neighbors and finds the one with the shortest distance. If it finds that this distance would be less (when adding it the current node's cost) then what its distanceFromSource is, then it updates itself and sets the flag to notify it's neighbors that its changed so they should recheck their values as well. The second section, simply sets each neighbor's shouldProcess flag to true.

This code could be considered the inner loop. The outer loop just keeps calling this kernel on all nodes until none of the kernels set the done flag to false.

Some Observations

The first and most important observation is that each node still gets the same cost as in the traditional Dijkstra's algorithm. This is important as it proves this method is at least calculating the cost to each node correctly. What is different however is the path generated by following the previous node path. Since the OpenCL version runs in parallel the path can change because the order nodes are updated can change. In each case this was observed the length of the path is identical, it's just a different path. In other words, this algorithm finds one of the shortest paths but not always the same shortest path.

Finally the number of iterations in the outer loop are also not constant. When running this on a fairly small map (16x12 or 192 nodes) the kernels were run between 18 and 21 times. On a much larger map (1024x768 or 786432 nodes) the number of iterations ran between x and y. When the number of iterations changed the path also changed, again demonstrating the semi random nature of parallel processors starting and finishing in an undetermined order.

The Results

The timing of this algorithm was not faster than traditional Dijkstra's in serial. However the time was generally close and only about twice as slow with larger maps. Smaller maps it was much slower due to the overhead in setting up the kernels. On our 1024x768 map, standard Dijkstra took about 0.7 second where this algorithm took around 1.5 seconds. Another interesting anomaly that I have not tracked down is an amount of time that is spend returning from the OpenCL cleanup process. My timing methods wrap the entire setup, processing, and teardown of the OpenCL code, but control I/O doesn't return for several seconds after the last timing is taken. This leads to reported timing far less than perceived timing. I assume this is due to the fact that using the GPUs is causing some kind of I/O blocking at some point, but as of yet I don't have a solid explanation as to what is causing this delay.

None the less, the results are promising. This is my first attempt with this algorithm and I am no expert at OpenCL. I believe with some optimizations both at the code level and the algorithm level, that this parallel version could give the serial version a run for its money. Two major places that I believe could be improved is not running kernels that have no node to be processed on and not returning back to the CPU on each iteration. The first issue could be resolved by using a set to track what nodes need updating. However that set would need to be usable in parallel and be

implemented in such a way as to not cause too much extra overhead on the global memory buffers. The second just requires more knowledge of OpenCL than I have. But likely setting up a control kernel in the GPU that triggers other kernels would be a drastic speed improvement over returning to the CPU and having it re-queue all the kernels for the next iteration.

Conclusion

Looking back on what we've learned.

PART SEVEN

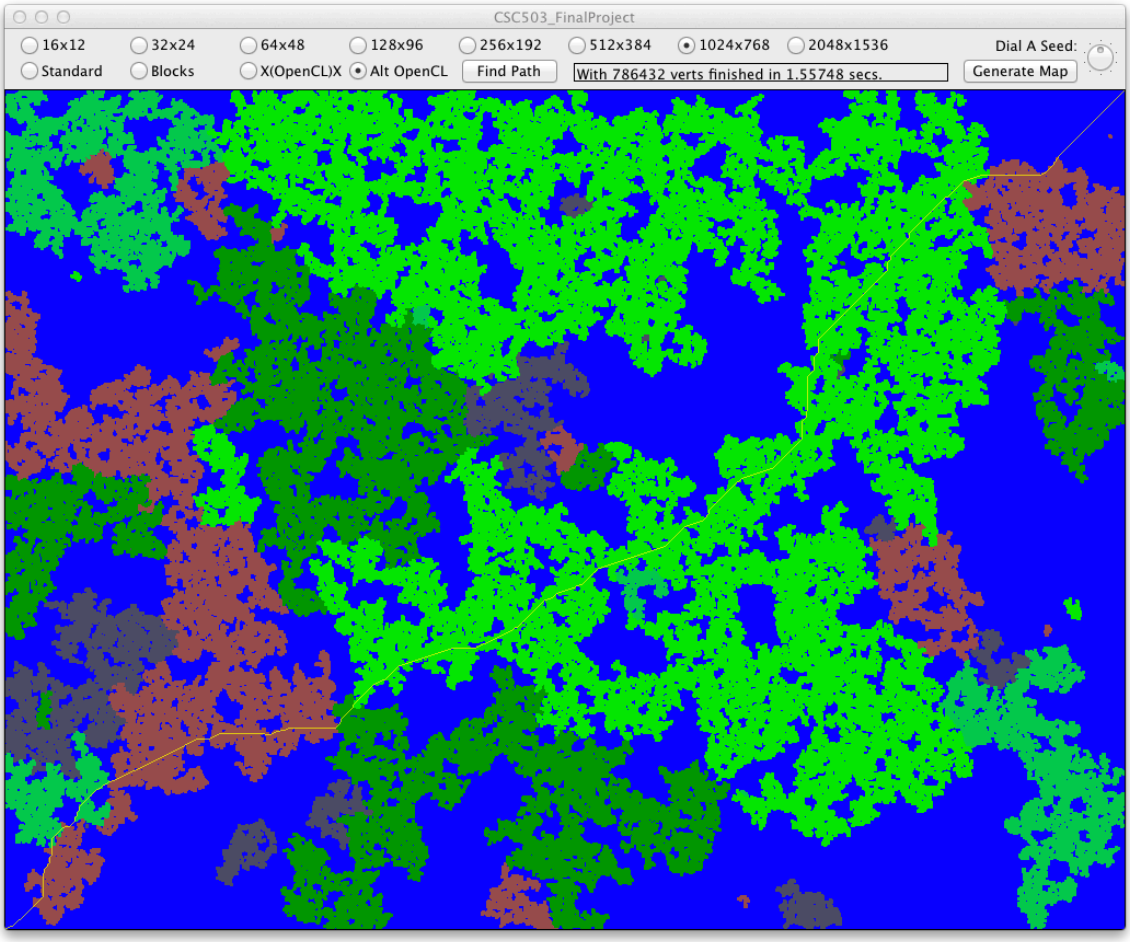
Although I was unable to create a parallel version of Dijkstra's algorithm that performed faster than the typical serial version, the processes used to parallelize algorithms were used and observations made that will help me think of solving problems in a new way. Parallel processors aren't going away, in fact we're sure to see a huge jump in the number of processors we have access to in the very near future (especially as things like OpenCL and other GPU languages become more and more supported). It's not even unreasonable to believe that typical CPUs could be designed more like modern day graphics cards and we'll go from seeing dual or quad or CPUs and start seeing kCore (that's 1024 cores!) CPUs. By changing the way we engineer our algorithms now, we'll be able to take advantage of these advances in CPU and GPU technologies. And we can start now. OpenCL is certainly one place, but even Obj-C's Blocks and GCD implementations are designed to be future compatible. Today we can only expect 2-4 blocks to be truly run in parallel but if the Mac OSX kernel has more cores to work with, more blocks would be run in parallel. An algorithm that takes advantage of parallel processing would truly become faster with more cores. Unlike most algorithms today that actually become slower with more cores since the clock speeds of multicore chips tend to be less than those of single core chips. Since the algorithms can only take advantage of the speed of a single thread, it simply can't get any faster with the way chips are being designed today.

The Code

All of the code for this project is on github at the following address https://github.com/steeleCSC503/CSC503_FinalProject. The application (and this paper) are also checked in there. You'll need a Mac with OSX 10.7 or better to run the applications since it takes advantage of program features that didn't exist prior. I used XCode 4.3 as the compiler for this project and the project files are included with the code.

The Application

This is a screen shot of the application that was written for this project and what each item does.



The grid sizes across the top allow you to choose the number of nodes in the map. Under that is the algorithm version that can be run. The basic OpenCL algorithm is by far the least efficient and can take upwards of minutes to return even on a tiny map, it's not recommend you use that option. This is most likely due to bad OpenCL coding than anything else however. The Alt OpenCL method is the algorithm used in the last section of this paper.

To use the app, select a map size, dial a seed (there are 10 maps of each size) and click "Generate Map". Then pick an algorithm and click "Find Path". At the end of each "Find Path" the timing results will print in the box to the right of the button. Again note that the Alt OpenCL path is found much faster than it reports (at least on my machine). Enjoy!