High Performance Computing

Coursework 5 – Sautrik Banerjee, Adi Krpo

User_Logic_sim.hpp

The 'Execute' function of this program, calls the function 'next', which contains a 'for' loop and also calls the recursive function 'CalcSrc'. We first attempted to improve code performance by making the following 'for' loop in the 'execute' function parallelisable –

```
for(unsigned i=0; i<pInput->clockCycles; i++){
    log->LogVerbose("Starting iteration %d of %d\n", i, pInput->clockCycles);
    state=next(state, pInput);
    log->Log(Log_Debug,[&](std::ostream &dst) {
            for(unsigned i=0; i<state.size(); i++){
                    dst<<state[i];
            }
    });
}
```

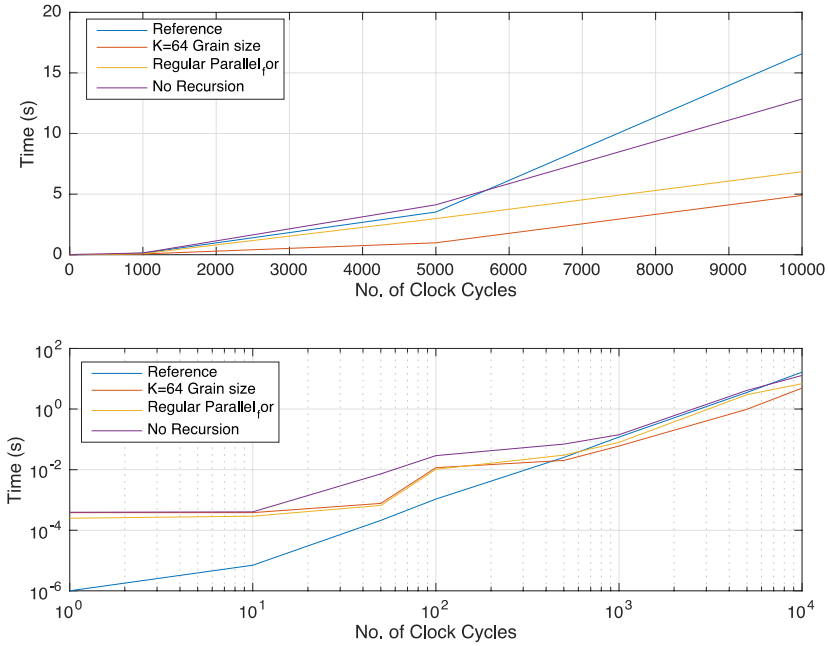
Since the vector 'state' was a write dependency, we decided to create a two dimensional vector of 'Boolean' type, to store different instances of 'state' i.e. the central function would be state[i+1] = next(state[i], pInput). However, this resulted in unsolvable memory allocation issues and no significant speed up.

We next decided to parallelise the 'for' loop in the 'next' function. This improved speed by a factor of 4. However, we were further able to exploit the parallelism of the loop by implementing adjustable grain sizes. After testing different grain sizes, we discovered a grain size of 64 improved performance the most.

We also attempted to convert the recursion in 'CalcSrc' to iteration. We successfully managed to do so, and the code is provided below-

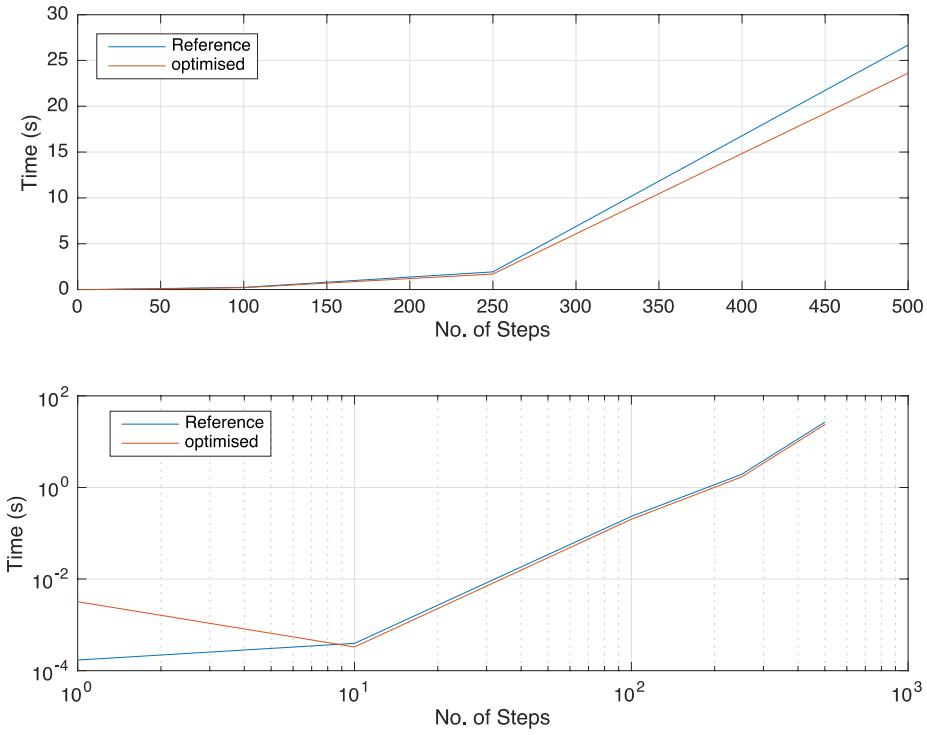
```
std::stack<unsigned> srcstack;
double a = 0;
double done = 0;
while(!done){
        if(src>state.size()){
          srcstack.push(src);
          src = input->xorGateInputs.at(src - state.size()).first;
        }else{
          a = a != state.at(src);
          if(!srcstack.empty()){
            src = srcstack.top();
            srcstack.pop();
            src = input->xorGateInputs.at(src-state.size()).second;
          }
          else{
            done = 1;
          }
        }
      }
      return a;
```

However, this resulted in performance improvement in rare cases. A graphical analysis of the program's progression in performance is given below.



User_Ising_spin.hpp

We began optimising this program by removing calls to unnecessary functions such as the 'lcg' and the 'count' functions. The operations being carried out by these functions can be done within the calling loop as they are simple mathematical operations. A future improvement could be the unrolling of the accumulate function, as it is likely to reduce execution. We also parallelised all loops without write dependencies



User_julia.hpp

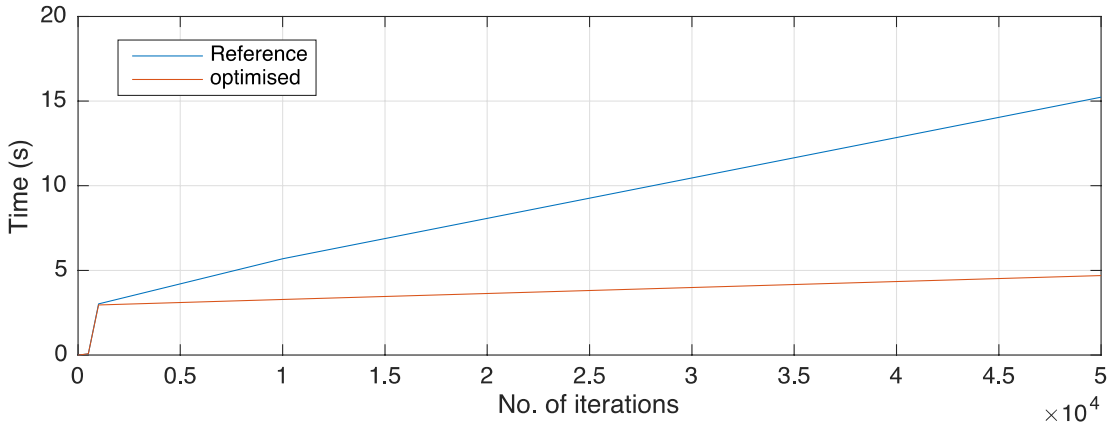
The Julia script was optimised using both chunking and an implementation in OpenCL. The chunk size was tested from 2 to 128 and K=16 was found to give the shortest run time the script.

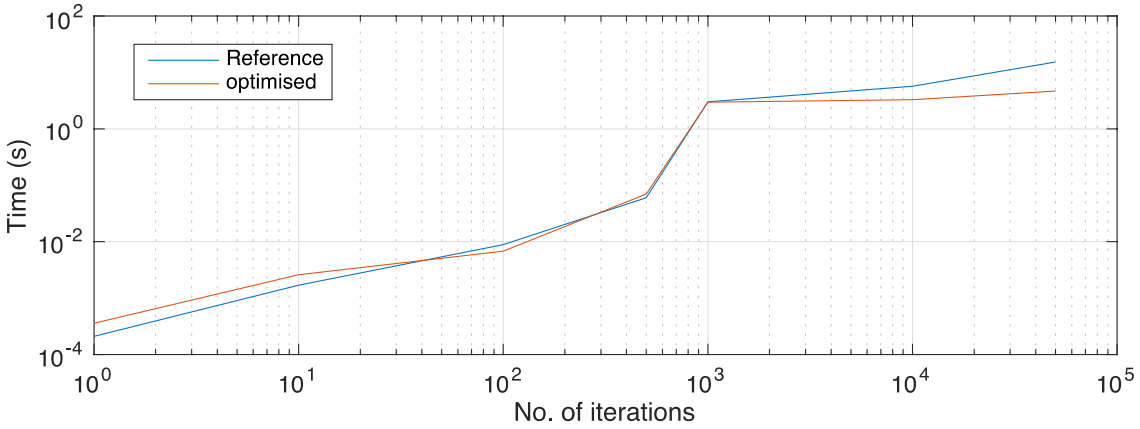
```
auto f = [=](const my_range_t &chunk)
   {
    for(unsigned y = chunk.begin(); y!=chunk.end();y++){
     for(unsigned x=0; x<width; x++) {
         // Map pixel to z_0
      puzzler::complex_t z(-1.5f+x*dx, -1.5f+y*dy);

         //Perform a julia iteration starting at the point z_0, for offset c.
         //   z_{i+1} = z_{i}^2 + c
         // The point escapes for the first i where |z_{i}| > 2.
      unsigned iter=0;
      while(iter<maxIter){
       if(abs(z) > 2){
        break;
       }
           // Anybody want to refine/tighten this?
       z = z*z + c;
       ++iter;
      }
      pDest[y*width+x] = iter;
     }
    }
   };

   tbb::parallel_for (range, f, tbb::simple_partitioner());
```

OpenCL was implemented by passing int height, int width, int maxIter, float2 c, __global uint *pDest and iterating over width and height and returning pDest through a buffer. In order to work with complex numbers in OpenCL, the following tutorial was used: http://stackoverflow.com/questions/10016125/complex-number-support-in-opencl.

```
__kernel void julia_kernel (int height, int width, int maxIter, float2 c, __global uint *pDest) {

        float dx=3.0f/width, dy=3.0f/height;

        int x = get_global_id(0);
        int y = get_global_id(1);
   //int loop3 = get_global_id(2);

        // typedef float2 cfloat;
        float2 z;

        z.x = -1.5f+x*dx;
        z.y = -1.5f+y*dy;

        int iter=0;
        while(iter<maxIter){
                if(cmod(z) > 2){
                        break;
                }
```

```
    // Anybody want to refine/tighten this?
                z = cmult(z,z)+ c;
                ++iter;
        }
        pDest[y*width+x] = iter;
};
```





User_random_walk.hpp

The random_walk function parameters were changed to include a vector for the count. The count vector was initialised as an atomic variable in order to allow multiple reads in the inner loop. The code is shown below:

```
 void random_walk(std::vector<puzzler::dd_node_t> &nodes,
std::vector<tbb::atomic<uint32_t>> &atomic_count, uint32_t seed, unsigned start, unsigned
length) const
 {
  uint32_t rng=seed;
  unsigned current=start;

  for(unsigned i=0; i<length; i++){

   atomic_count[current]++;

   unsigned edgeIndex = rng % nodes[current].edges.size();
   rng=rng*1664525+1013904223;
```

```
    current=nodes[current].edges[edgeIndex];
  }
 }
```

In the outer loop, the random number generator was removed from the outer for loop to allow the outer loop to be parallelized. We found that tbb:parallel_for worked fastest for this through trial and error.

The OpenCL implementation of random_walk

References:

In order to work with complex numbers using float2 in OpenCL, the following tutorial was used: http://stackoverflow.com/questions/10016125/complex-number-support-in-opencl, 23/11/2016