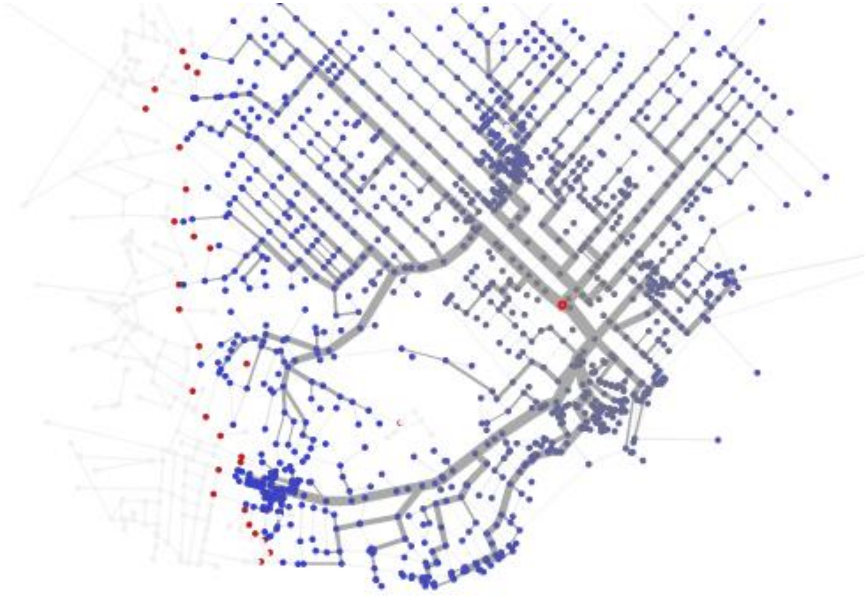
UCI Samueli School of Engineering

Dijkstra’s Algorithm OpenMP and CUDA Parallelization

Presented by:
Jingyuan Wang and Jianbo Pei

Outline

- Serial Dijkstra’s Algorithm Review
- OpenMP Parallelization
  - Approaches
  - Results
- CUDA Parallelization
  - Approaches
  - Results
- Conclusion - OpenMP vs CUDA
- Future Works



Serial Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm aiming at finding the shortest path between the source and other nodes in a graph.

Let V be a set of all the nodes, and S be a set of explored nodes.

– For each u in S we know the shortest path distance from s, d(u).

■ Initially S = {s}, d(s) = 0; for all v ≠ s, d(v) = ∞

■ While S≠V

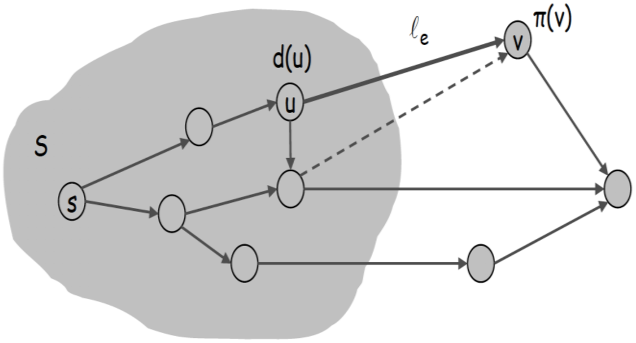
– Select unexplored node v (in V-S) that minimizes "distance label"

$$\pi(v) = \min_{e=(u,v):u\in S} (d(u) + \ell_e)$$

– Add v to S and set d(v) = π(v)

– Update distance label π(w) for all neighbors w of v

■ EndWhile



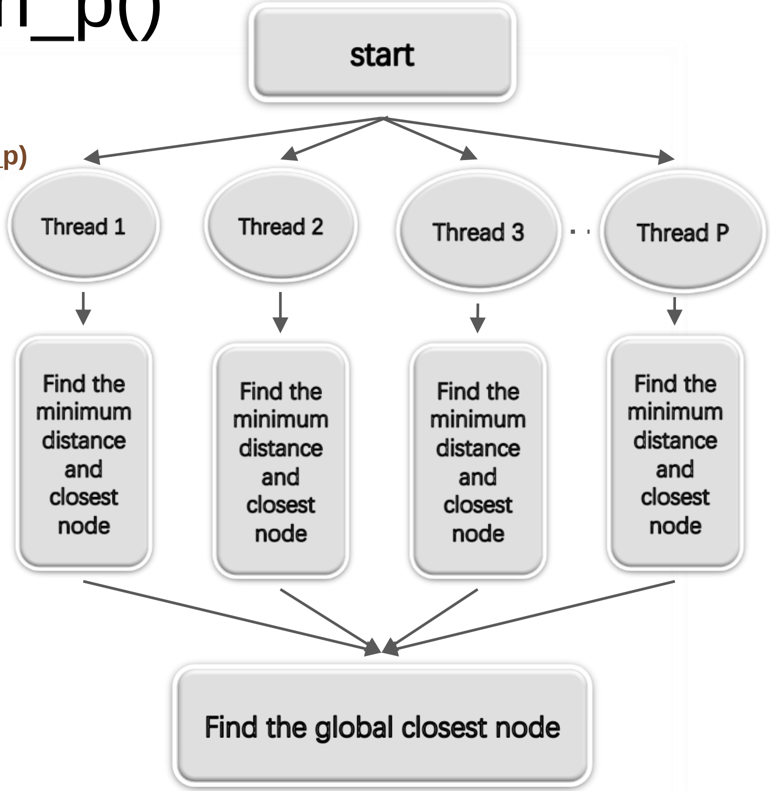
Two functions to parallel.

OpenMP Parallelization - findMin_p()

```
int findMin_p(int *dist_p, bool *visit_p){
omp_set_num_threads(THREADS);
#pragma omp parallel private(min_dist_thread, min_node_thread) shared(dist_p, visit_p)
   {
       min_dist_thread = min
       min_node_thread = minNode
#pragma omp barrier
#pragma omp for nowait
       for vertex = 0 to N with increment 4 {
          if ((dist_p[vertex] < min_dist_thread) && (visit_p[vertex] == false))
             Update min_dist_thread and min_node_thread
                         :

       }
#pragma omp critical
       {
          if (min_dist_thread < min)
             Update global closest node
       }
   }
   return minNode;
}
```

Unroll four iterations of for loop.



OpenMP Parallelization - updateDist_p()

```
visit_p[minNode] = true;
void updateDist_p(){
        omp_set_num_threads(THREADS);
#pragma omp parallel
        {
#pragma omp for
            for vertex = 0 to N with increment 4 {
                if (the vertex has not been visited && the vertex connects to the minNode && distance[vertex] >
distance[minNode] + graph[minNode][vertex])
                    distance[vertex] = distance[minNode] + graph[minNode][vertex];
                                    ⋮

            }
#pragma omp barrier
        }
}
```

Unroll four iterations of for loop.

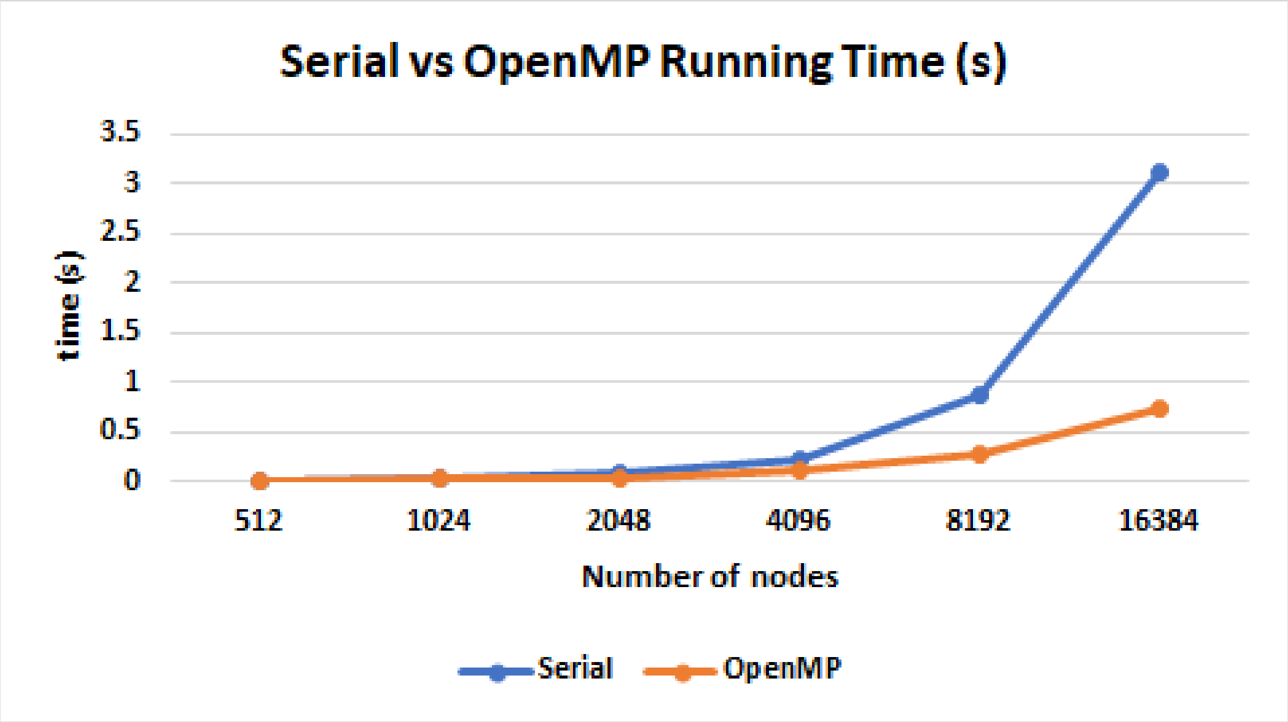
UCI Samueli School of Engineering

OpenMP - Results

Number of nodes	512	1024	2048	4096	8192	16384
Serial	0.005327	0.021351	0.084384	0.219032	0.867634	3.12067
OpenMP	0.010836	0.022873	0.045601	0.111297	0.286892	0.745239
Number of Threads	8	8	8	8	8	8

Highest improvement:
3.2x faster than serial

OpenMP - Results



CUDA Parallelization - Environment

- GPU: Tesla M2090 (512 cores, 6GB memory)
- Compile operations:
  - module load cuda/5.0
  - module load gcc/4.4.3
  - nvcc -arch=sm_11 dijkstra_cuda.cu -o dijkstra_cuda

CUDA Parallelization - closestNodeCUDA( )

Step 1:

```
__global__ void closestNodeCUDA(int* min_value, int* minIndex, int* temp,

    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int cache[THREADS_PER_BLOCK];
    __shared__ int cacheIndex[THREADS_PER_BLOCK];


    if (index < num_vertices) {
        if ((node_dist[index]) < INT_MAX && (visited_node[index]) == 0) {
            cache[threadIdx.x] = node_dist[index];
            cacheIndex[threadIdx.x] = index;
        }
        else {
            cache[threadIdx.x] = INT_MAX;
            cacheIndex[threadIdx.x] = -1;
        }
    }
    __syncthreads();
```

__shared__:

cache[ ]: load distance values from global node_dist array
cacheIndex[ ]: store corresponding indices

CUDA Parallelization - closestNodeCUDA( )

Step 2:

```
unsigned int i = blockDim.x / 2;
while (i != 0) {
    if (threadIdx.x < i) {
        if (cache[threadIdx.x + i] < cache[threadIdx.x]) {
            cache[threadIdx.x] = cache[threadIdx.x + i];
            cacheIndex[threadIdx.x] = cacheIndex[threadIdx.x + i];
        }
    }
    __syncthreads();
    i /= 2;
}

if (threadIdx.x == 0) {
    temp[blockIdx.x] = cache[0];
    tempIndex[blockIdx.x] = cacheIndex[0];
}
```

reduction to find min and its index for each block.

store them at cache[0] and cacheIndex[0]

global memory:

temp[ ]: store all min from each block
tempIndex[ ]: store corresponding indices

CUDA Parallelization - closestNodeCUDA( )

Step 3:

```
unsigned int k = BLOCKS / 2;
if (threadIdx.x == 0 && blockIdx.x == 0) {
    while (k != 0) {
        for (int j = 0; j < k; ++j) {
            if ((temp[j + k]) < temp[j]) {
                temp[j] = temp[j + k];
                tempIndex[j] = tempIndex[j + k];
            }
        }

        __syncthreads();
        k /= 2;
    }
}
```

reduction to find min and minIndex from global temp[ ] and global tempIndex[ ] using only thread 0 at block 0.

store min at temp[0]
store minIndex at tempIndex[0]

CUDA Parallelization - closestNodeCUDA( )

Step 4:

```
if (threadIdx.x == 0 && blockIdx.x == 0) {
    *min_value = temp[0];
    *minIndex = tempIndex[0];

    global_closest[0] = *minIndex;
    visited_node[*minIndex] = 1;
}
__syncthreads();
```

finally, return min value from temp[0] and the corresponding index from tempIndex[0].

store minIndex to global_closest[0] which will be used in cudaRelax( ) function

mark this node as visited

CUDA Parallelization - cudaRelax( )

```
__global__ void cudaRelax(data_t* graph, data_t* node_dist,
    int next = blockIdx.x * blockDim.x + threadIdx.x;
    int source = global_closest[0];

    data_t edge = graph[source * VERTICES + next];
    data_t new_dist = node_dist[source] + edge;

    if ((edge != 0) &&
        (visited_node[next] != 1) &&
        (new_dist < node_dist[next])) {
        node_dist[next] = new_dist;
    }

}
```

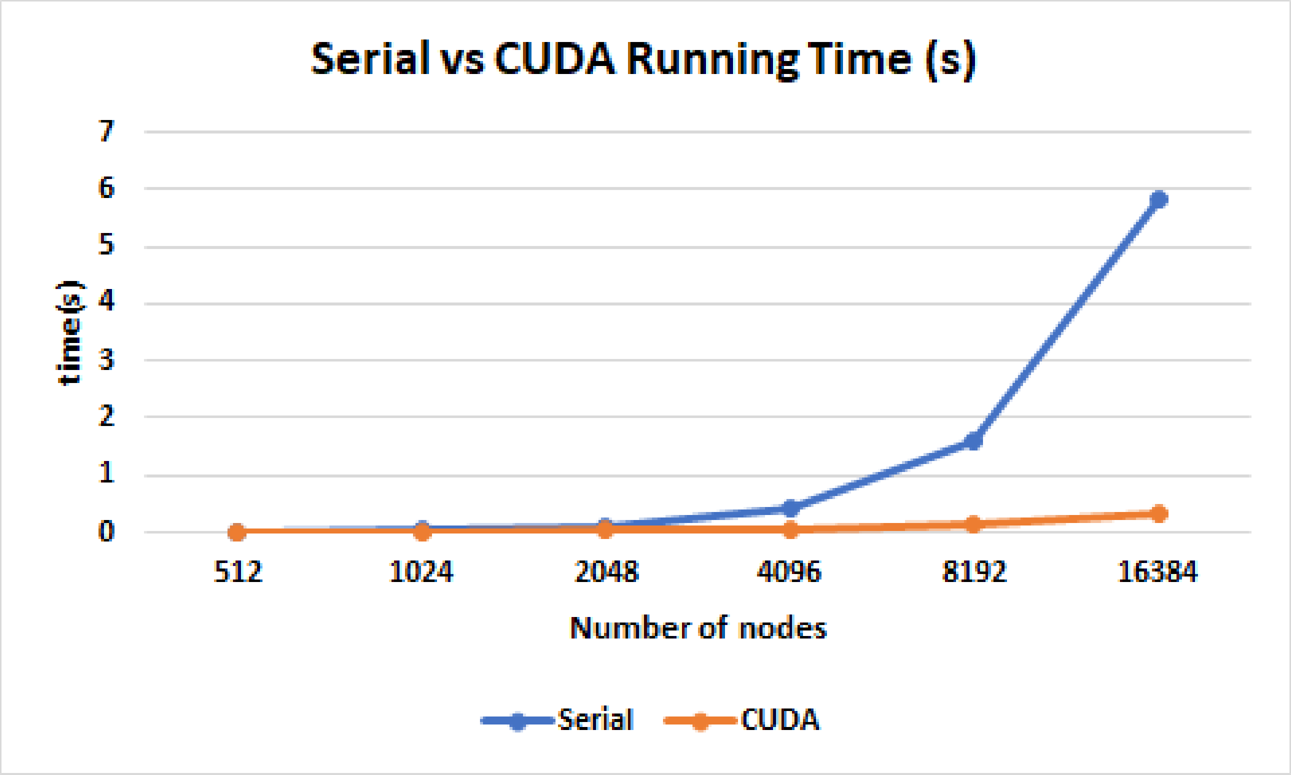
Update all nodes distance that are connected with global_closest node, which is returned from last function.

CUDA - Results

Number of nodes	512	1024	2048	4096	8192	16384
Serial	0.01	0.03	0.09	0.4	1.58	5.81
CUDA	0.005334	0.010972	0.023383	0.051648	0.119978	0.330931
Blocks	2	2	4	8	16	32
Threads/block	256	512	512	512	512	512

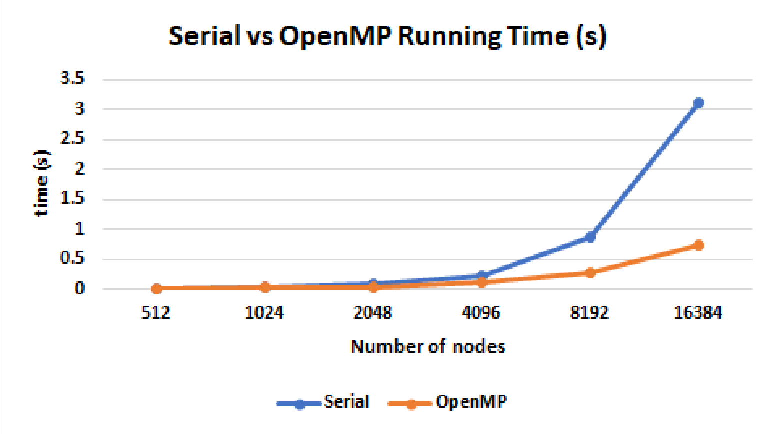
Highest improvement:
16.6x faster than serial

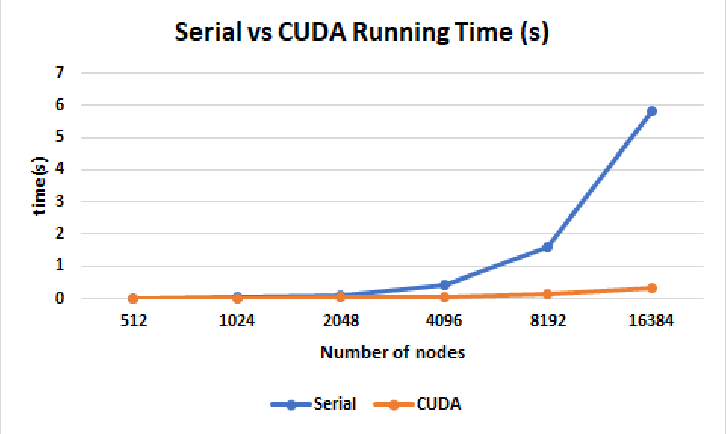
CUDA - Results



Conclusion - OpenMP vs CUDA

- In theory, running time complexity:
  - Serial: O(N^2)
  - OpenMP: O(N * N/P)
  - CUDA: O(N* log(N))
- In our experiments, CUDA and OpenMP behave quite similar as above.





Future Works

- Looking for more optimization methods to improve current work.
- Testing on much larger graphs and real life graphs.

Thank you!

Questions?