Task Graph Programming using DOCA Graph

Dian-Lun Lin Cheng-Hsiang Chiu

Takeaways

- Understand the motivation behind task graph programming
- Learn to use the DOCA graph programming model
- Build an application based on DOCA graph
- Evaluate DOCA graph on microbenchmarks
- Conclude the talk

Background

- We are from a research group targeting on parallel and heterogeneous computing
  - Special focus on task graph parallelism (i.e., task graph programming)

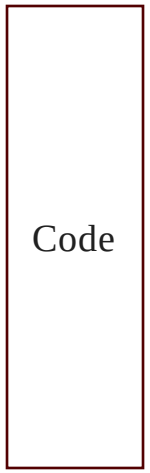
DOCA graph has similar idea

Takeaways

- Understand the motivation behind task graph programming
- Learn to use the DOCA graph programming model
- Build an application based on DOCA graph
- Evaluate DOCA graph on microbenchmarks
- Conclude the talk

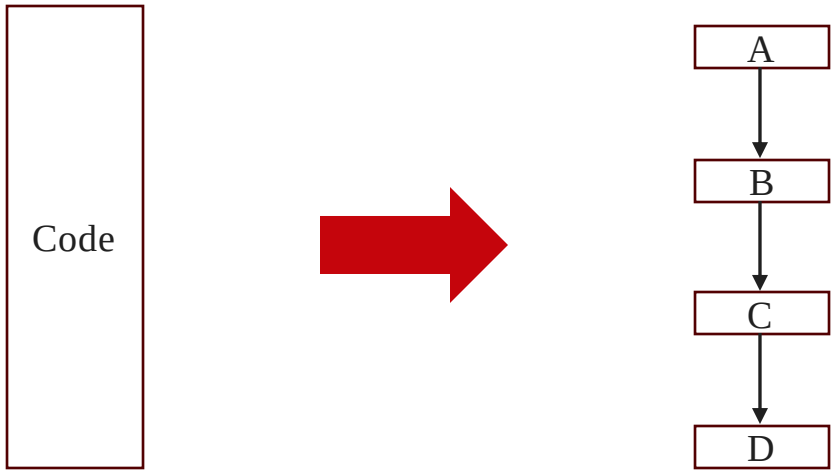
What is Task Graph-based Programming

- Task graph-based programming encapsulates function calls and their dependencies in a top-down task graph



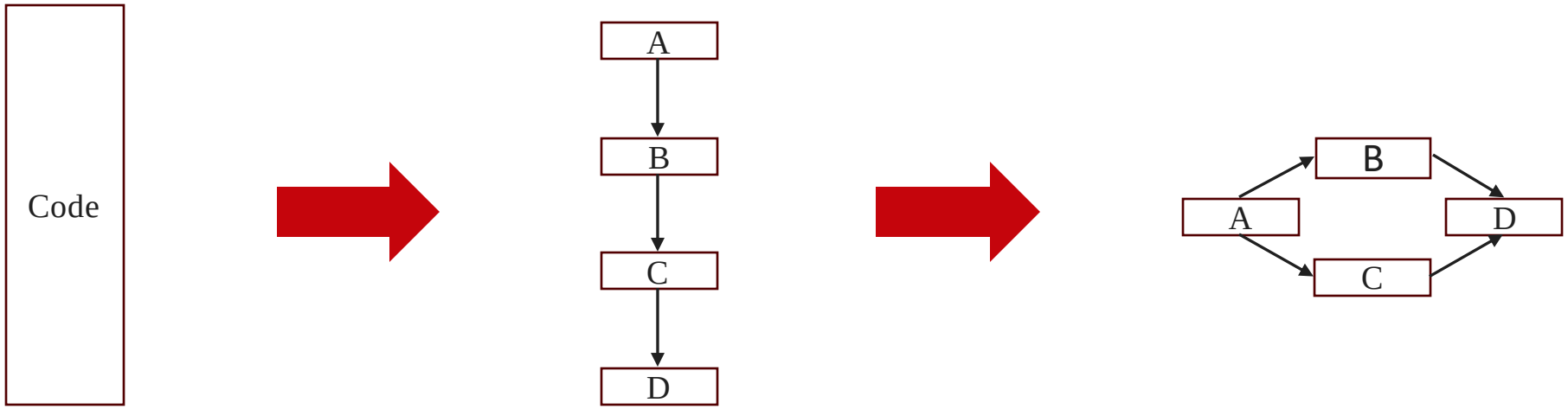
What is Task Graph-based Programming

- Task graph-based programming encapsulates function calls and their dependencies in a top-down task graph



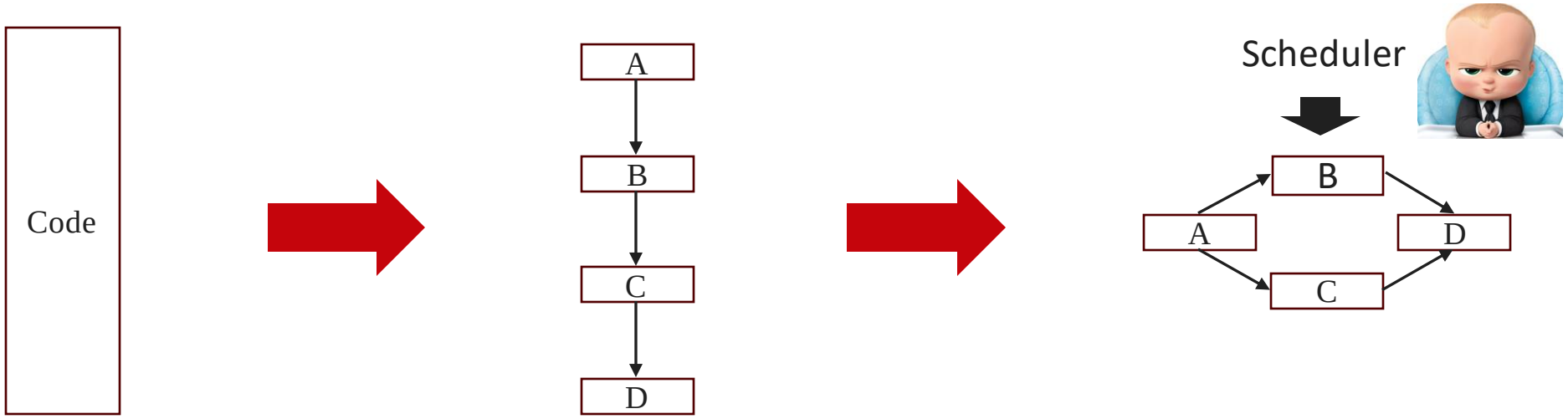
What is Task Graph-based Programming

- Task graph-based programming encapsulates function calls and their dependencies in a top-down task graph



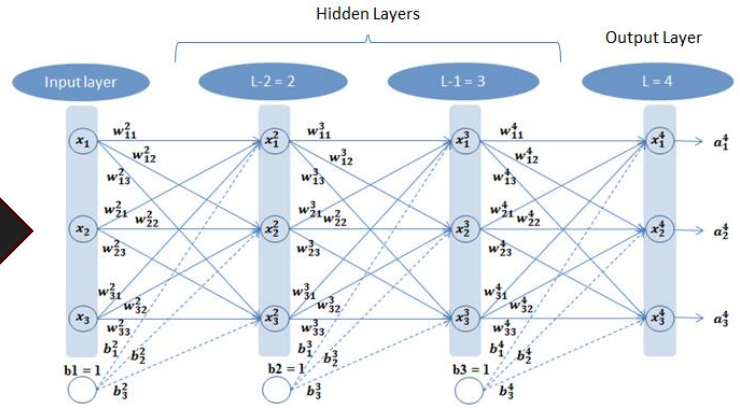
What is Task Graph-based Programming

- Task graph-based programming encapsulates function calls and their dependencies in a top-down task graph



GPU Example

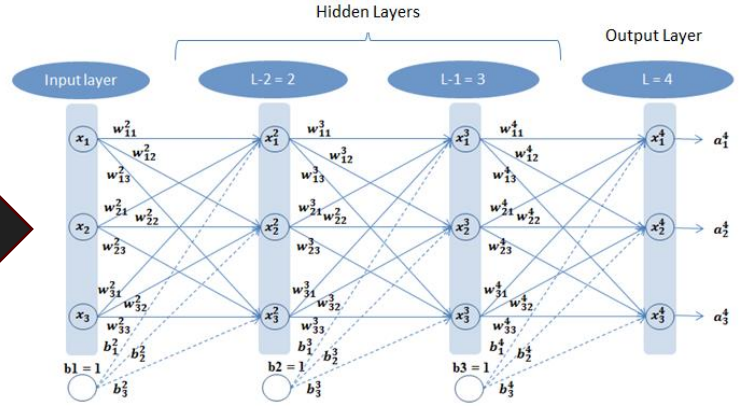
Grab an input batch



Identify the categories

GPU Example

Grab an input batch



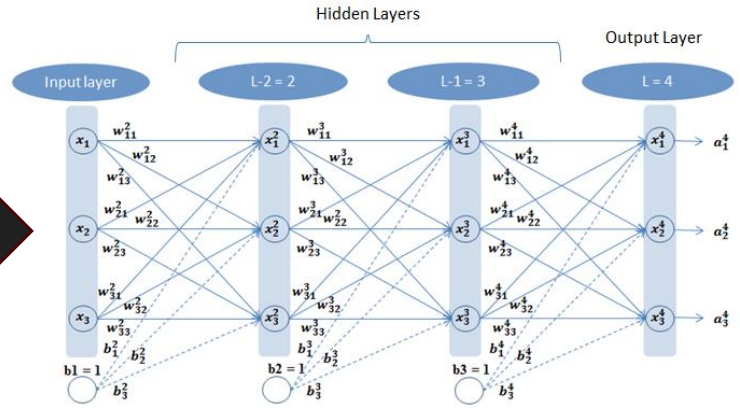
Identify the categories

Iteratively run many thousands of times to perform inference

Conventional approach: CUDA Stream

Grab an input batch

MNIST sparse 256x256



Identify the categories

Iteratively run many thousands of times to perform inference

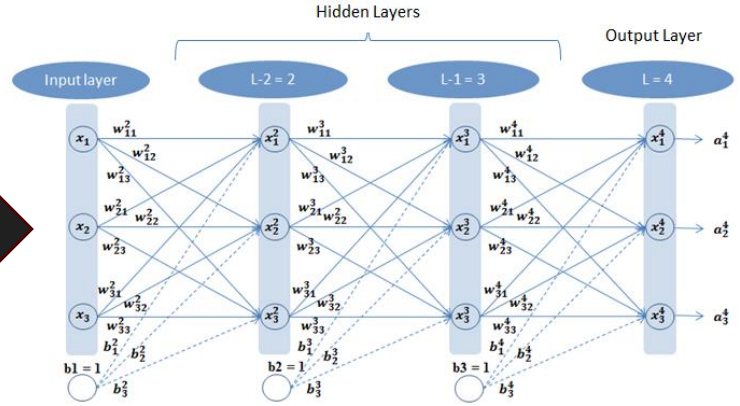
```
for(int batch=0; batch<60000; batch++){
  for(int layer=0; layer<1920; layer++){
    inference<<<grid, block, shm, stream>>>(out_d, in_d);
    cudaStreamSynchronize(stream);
  }
}
```

Use CUDA stream to enqueue the inference kernel

Stream can Incur Huge Overhead for Iterative Pattern

Grab an input batch

MNIST sparse 256x256



Identify the categories

Iteratively run many thousands of times to perform inference

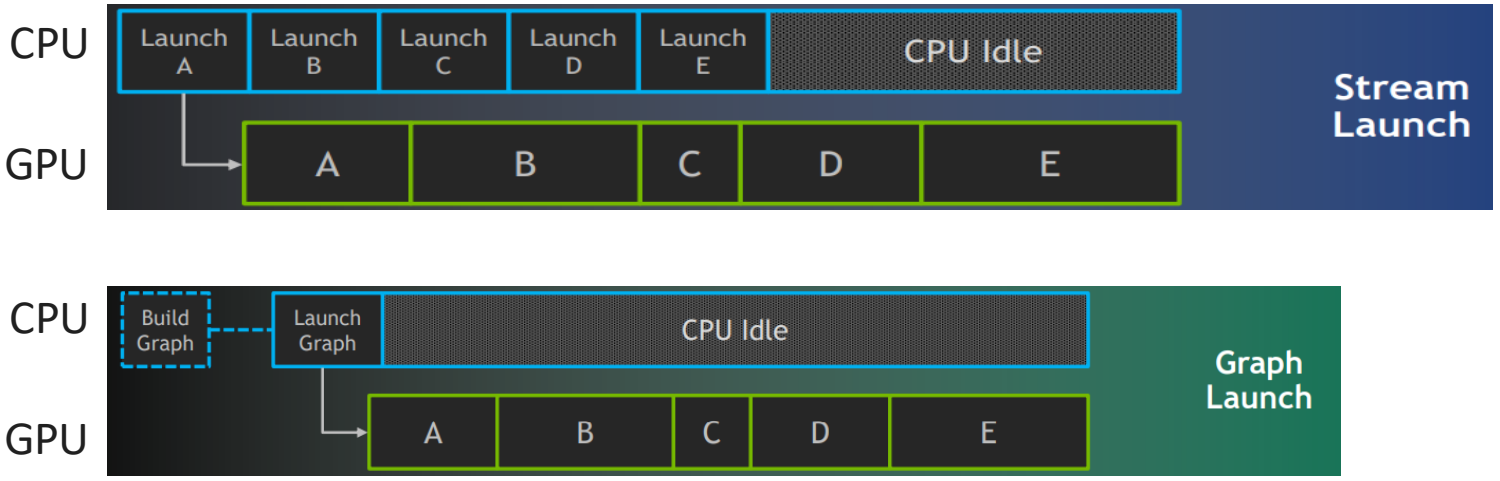
```
for(int batch=0; batch<60000; batch++){
  for(int layer=0; layer<1920; layer++){
    inference<<<grid, block, shm, stream>>>(out_d, in_d);
    cudaStreamSynchronize(stream);
  }
}
```

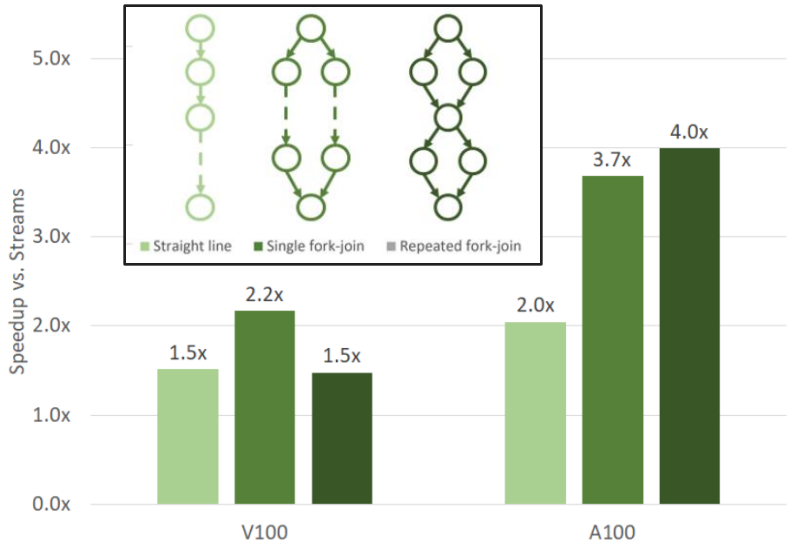
Execution overhead

Synchronization overhead

New execution model: CUDA Graph

- CUDA Graph removes stream launch overhead for iterative patterns
  - Launch a CUDA graph requires only a single CPU call
  - CUDA runtime can perform the whole-graph optimization
  - New GPU architectures (e.g., A100) have many task graph optimizations





Takeaways

- Understand the motivation behind task graph parallelism
- Learn to use the DOCA graph programming model
- Build an application based on DOCA graph
- Evaluate DOCA graph on microbenchmarks
- Conclude the talk

DOCA Graph Programming Model - Node

- Context node
  - DOCA job
- User node
  - User callback job
- Sub-graph node
  - An instance of another DOCA graph
  - DOCA Graph is composable

DOCA Graph Programming Model - APIs

- `doca_graph_create`
  - Create a DOCA graph
- `doca_graph_ctx_node_create`
  - Create a DOCA node
- `doca_graph_add_dependency`
  - Create a dependency between two DOCA nodes
- `doca_workq_graph_submit`
  - Submit the entire DOCA graph to work queue
- `doca_workq_progress_retrieve`
  - Iteratively call this function until return DOCA_SUCCESS

DOCA Graph Programming Model - Restrictions

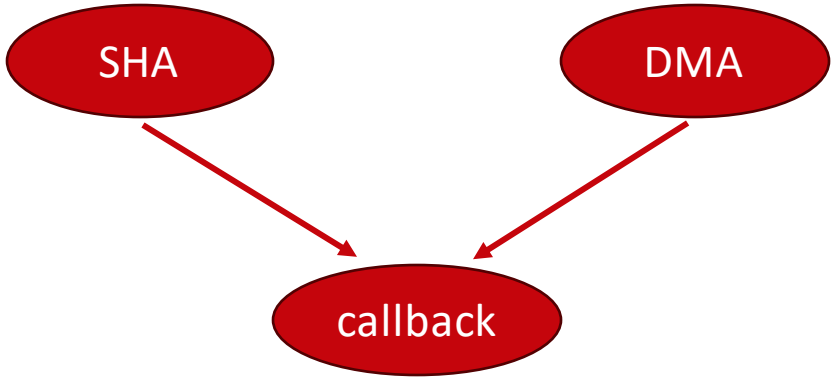
- Does not support circle dependencies
- A DOCA graph must contains at least one context node (i.e., DOCA job)
  - A sub-graph that contains a context node is allowed

Takeaways

- Understand the motivation behind task graph parallelism
- Learn to use the DOCA graph programming model
- Build an application based on DOCA graph
- Evaluate DOCA graph on microbenchmarks
- Conclude the talk

DOCA Graph Sample

- The sample calculates a SHA value and copies a source buffer to a destination buffer in parallel.
- The graph ends with a user callback node that prints the SHA value and compares the source with the DMA destination.



DOCA Graph Sample Flow - 1

- Create the graph using ***doca_graph_create***.
- Create the graph nodes using ***doca_graph_ctx_node_create***.
- Define dependencies using ***doca_graph_add_dependency***.
- Start the graph using ***doca_graph_start***.
- Add the graph to a work queue using ***doca_graph_workq_add***.

DOCA Graph Sample Flow - 2

- Create the graph instance using ***doca_graph_instance_create***.
- Set the nodes data using ***doca_graph_instance_set_ctx_node_data***.
- Submit the graph instance to the work queue using ***doca_workq_graph_submit***.
- Call ***doca_workq_progress_retrieve*** until it returns DOCA_SUCCESS.

DOCA Graph Sample Execution

```
Instance 7 user callback
SHA Value: 76dc13d83659a209fe0f516a18d82e0db47abbd96a1ab7dc65eca67455c3d7aa
Instance 7 completed
Instance 8 user callback
SHA Value: 666373a5c4cc310a18872ca337735981cd9764dac596b67839405983339768 1e
Instance 8 completed
Instance 9 user callback
SHA Value: 738c7aebba29b335a694d64478e480a60e74bcbe00b585063b4d290704e7e91a
Instance 9 completed
All instances completed successfully
```

Takeaways

- Understand the motivation behind task graph parallelism
- Learn to use the DOCA graph programming model
- Build an application based on DOCA graph
- Evaluate DOCA graph on microbenchmarks
- Conclude the talk

Evaluation Setup

- We studied the runtime performance between DOCA Graph, Pthread, and sequential implementation.
- We compiled the programs using gcc 11.4.
- We ran all the experiments on a Ubuntu Linux 22.04 host with an AMS EPYC 7302 16-Core CPU at 128 GB RAM and a BlueField-2 DPU.
- We ran every implementation up to 500 instances and measured the runtime using /usr/bin/time.

Baseline – Sequential Implementation

- We spawned only one thread.

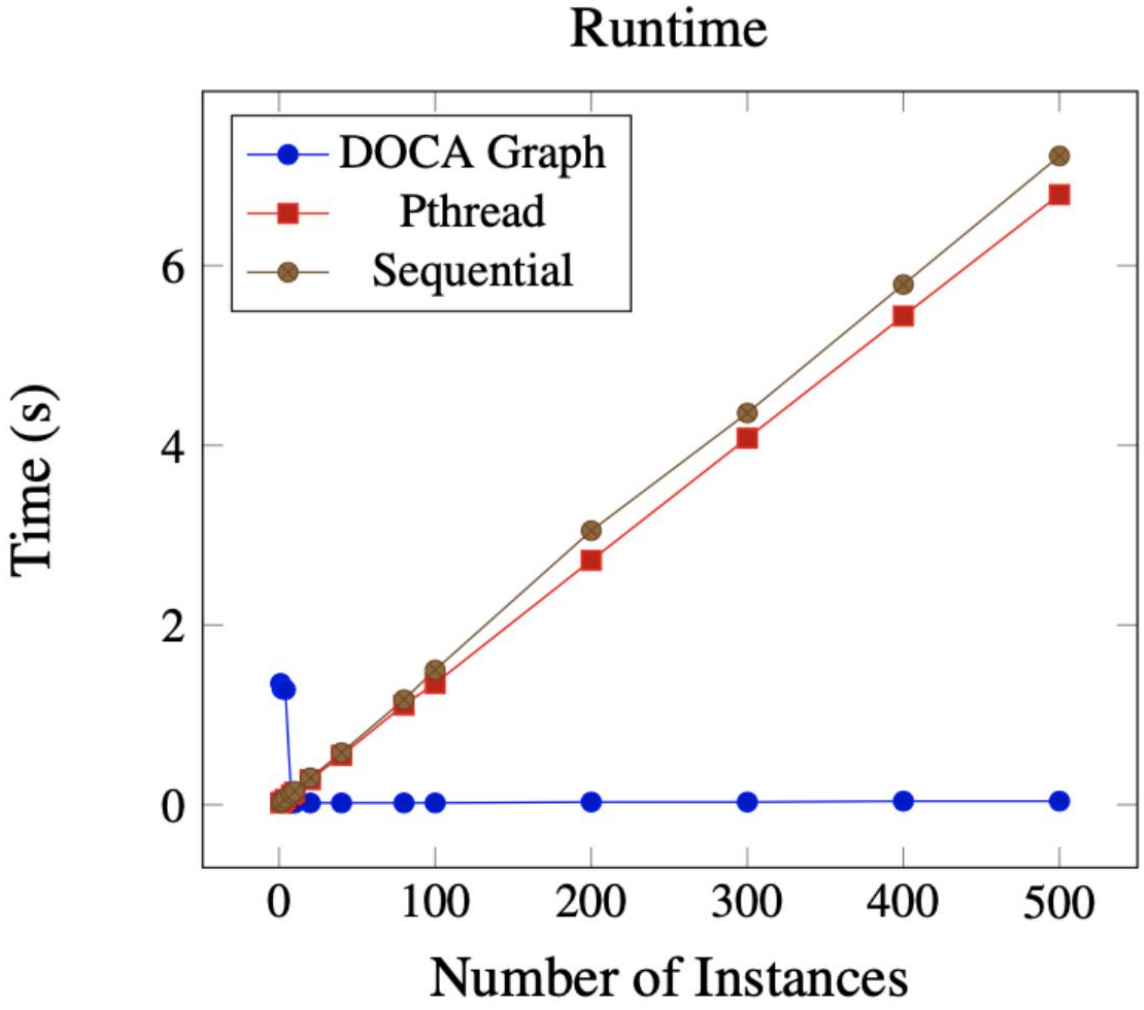
```
1  int instances = 500;
2
3  pthread thread;
4  for (int i = 0; i < instances; ++i) {
5    // Run SHA
6    pthread_create( & thread, NULL, run_sha, NULL);
7    pthread_join(thread, NULL);
8
9    // Run DMA
10   pthread_create( & thread, NULL, run_dma, NULL);
11   pthread_join(thread, NULL);
12 }
```

Baseline – Pthread Implementation

- We spawned two threads and explicitly joined them after one instance.

```
1  int instances = 500;
2
3  pthread_t thread_sha, thread_dma;
4
5  for (int i = 0; i < instances; ++i) {
6    // Spwan the threads
7    pthread_create( & thread_sha, NULL, run_sha, NULL);
8    pthread_create( & thread_dma, NULL, run_dma, NULL);
9
10   // Join the threads
11   pthread_join(thread_sha, NULL);
12   pthread_join(thread_dma, NULL);
13 }
```

Runtime Comparison

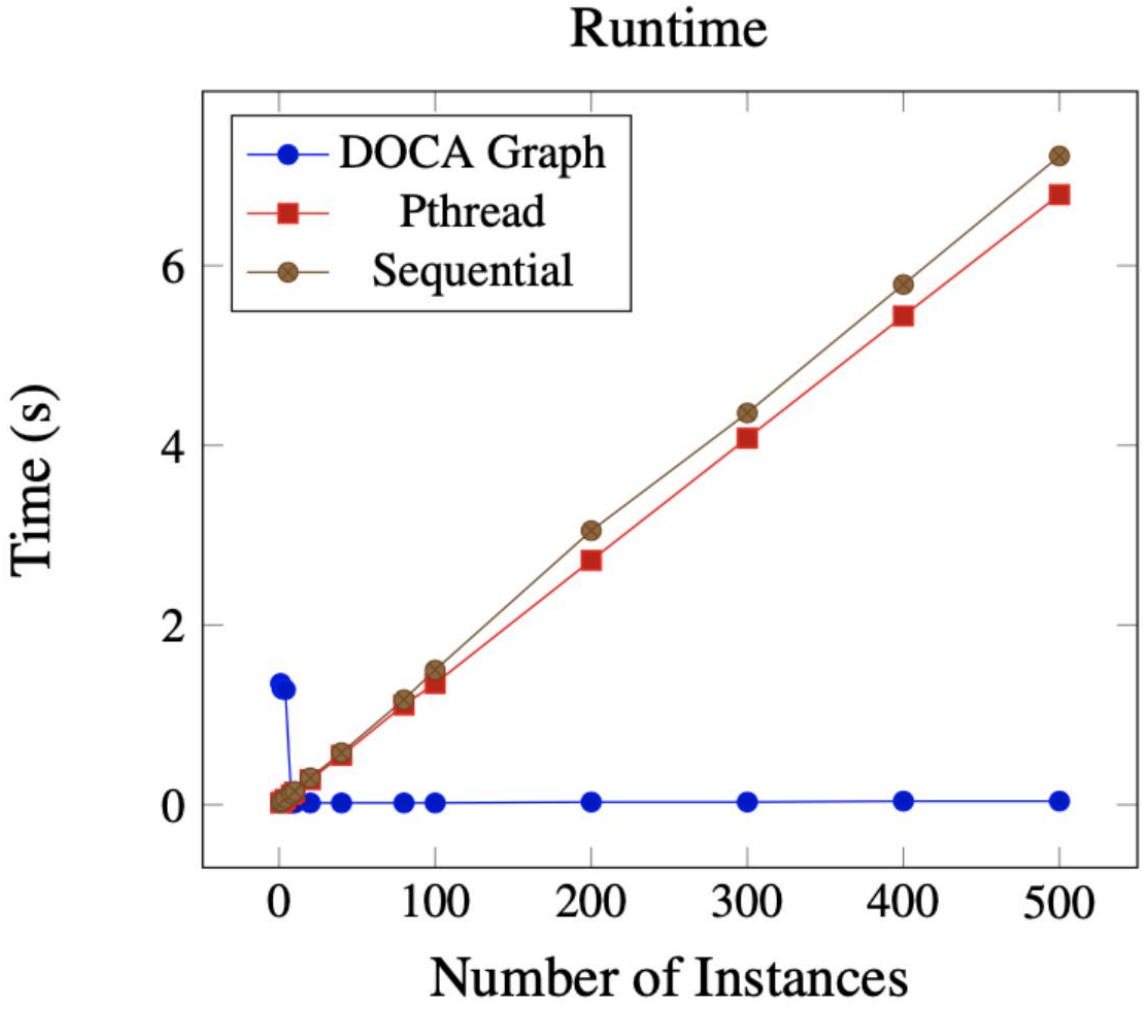


Horizontal axis denotes the number of instances.

Vertical axis denotes the runtime in seconds.

Blue color is DOCA Graph code.
Red color is Pthread code.
Soil color is sequential code.

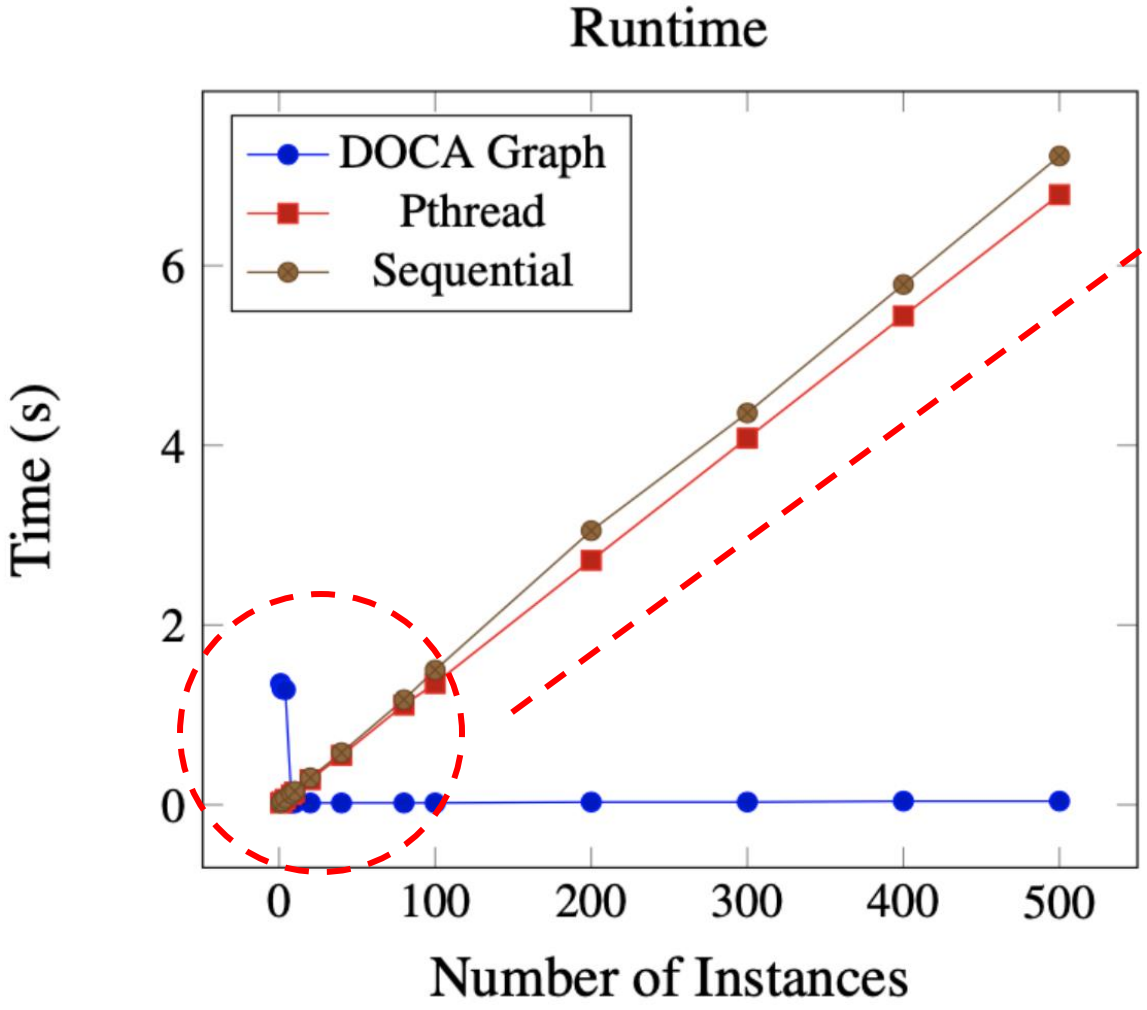
Runtime Comparison



Pthread is consistently faster than sequential code.

Because Pthread code runs the two tasks, SHA and DMA, in parallel.

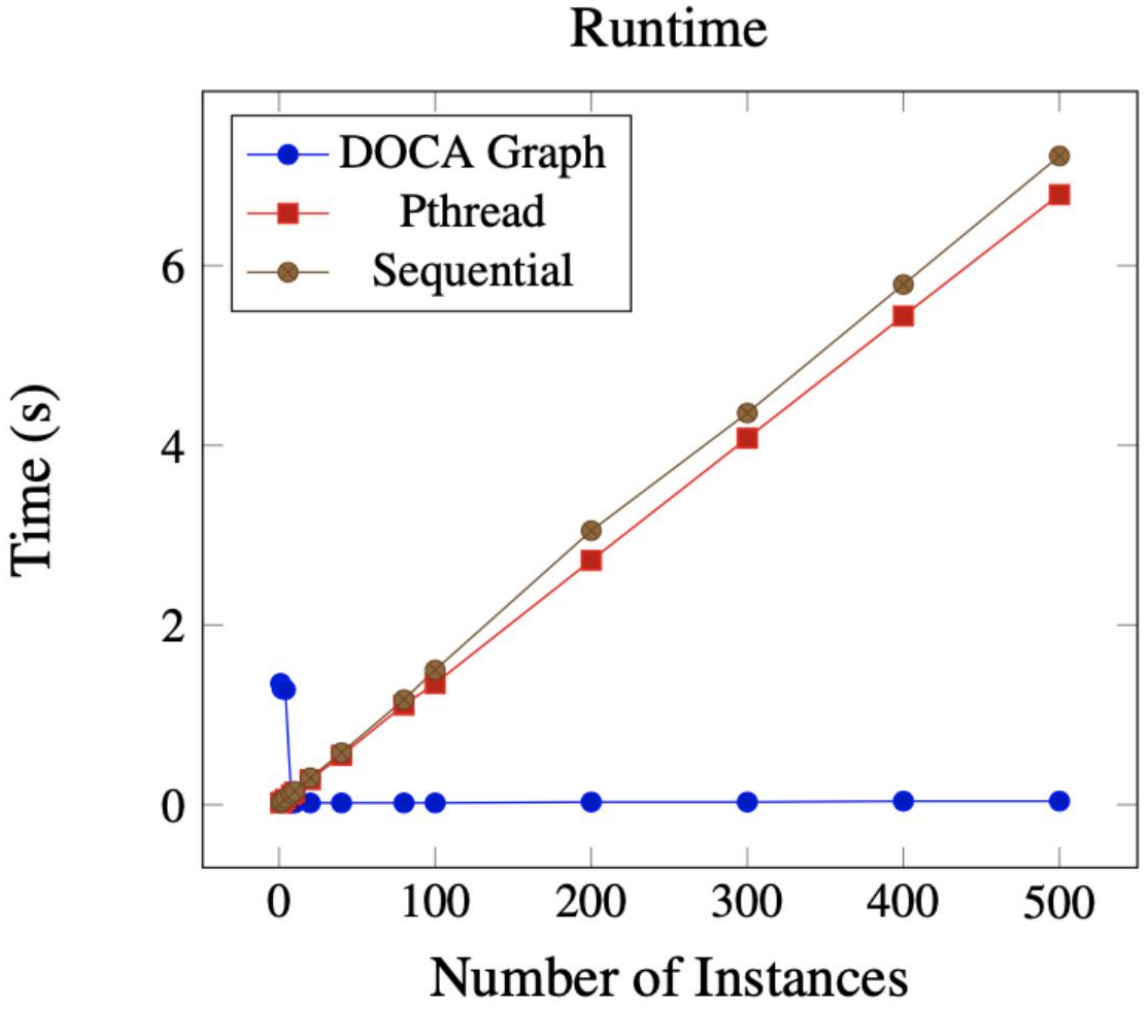
Runtime Comparison



For small instances (< 5), DOCA Graph is much slower than Pthread and sequential code.

Because building the graph has overhead for DOCA Graph.

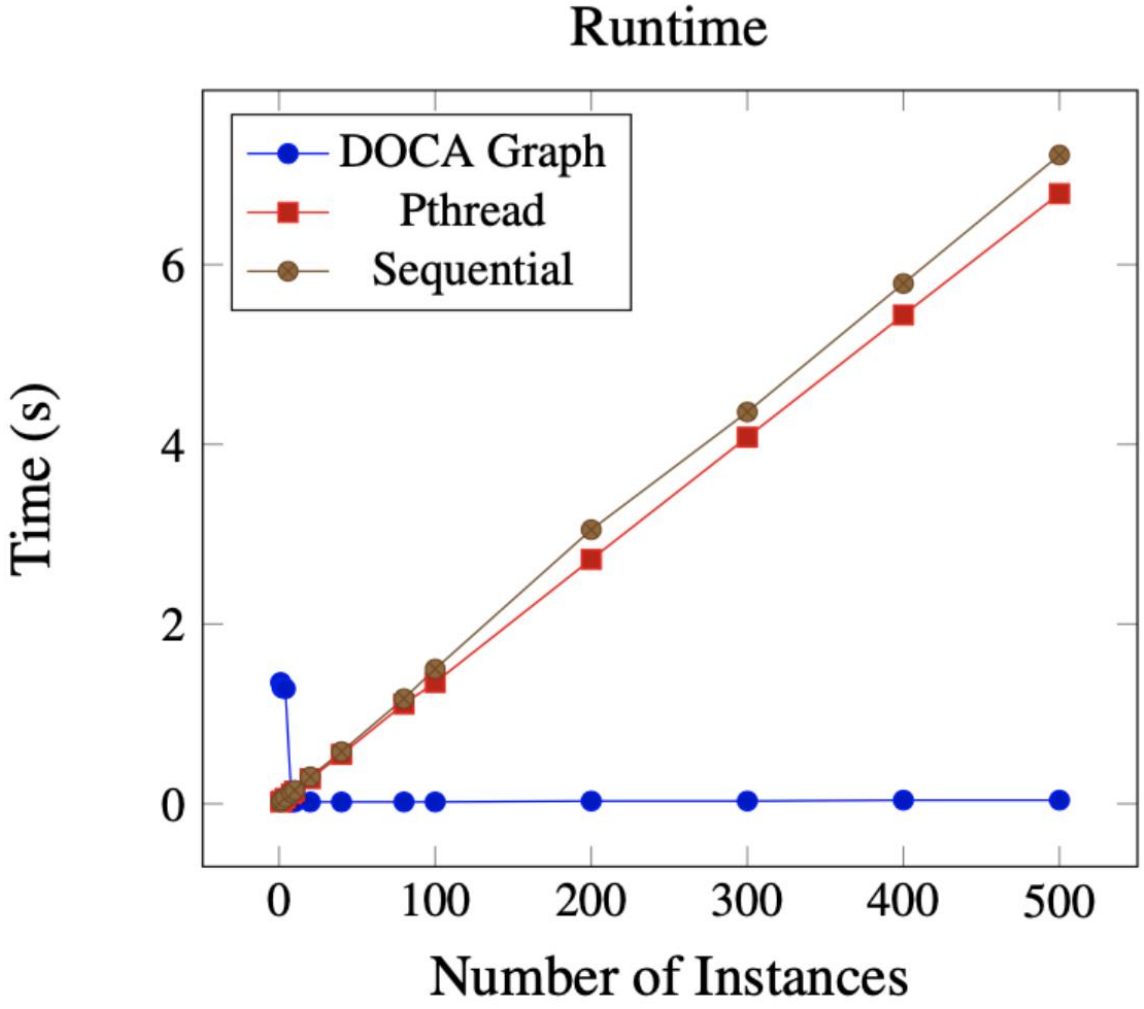
Runtime Comparison



For big instances (> 5), DOCA Graph is much faster than Pthread and sequential code.

Because Pthread needs synchronizations after every instance. (pthread.join is not free). From our experience, DOCA Graph could use lightweight atomic counter to do the synchronization.

Runtime Comparison



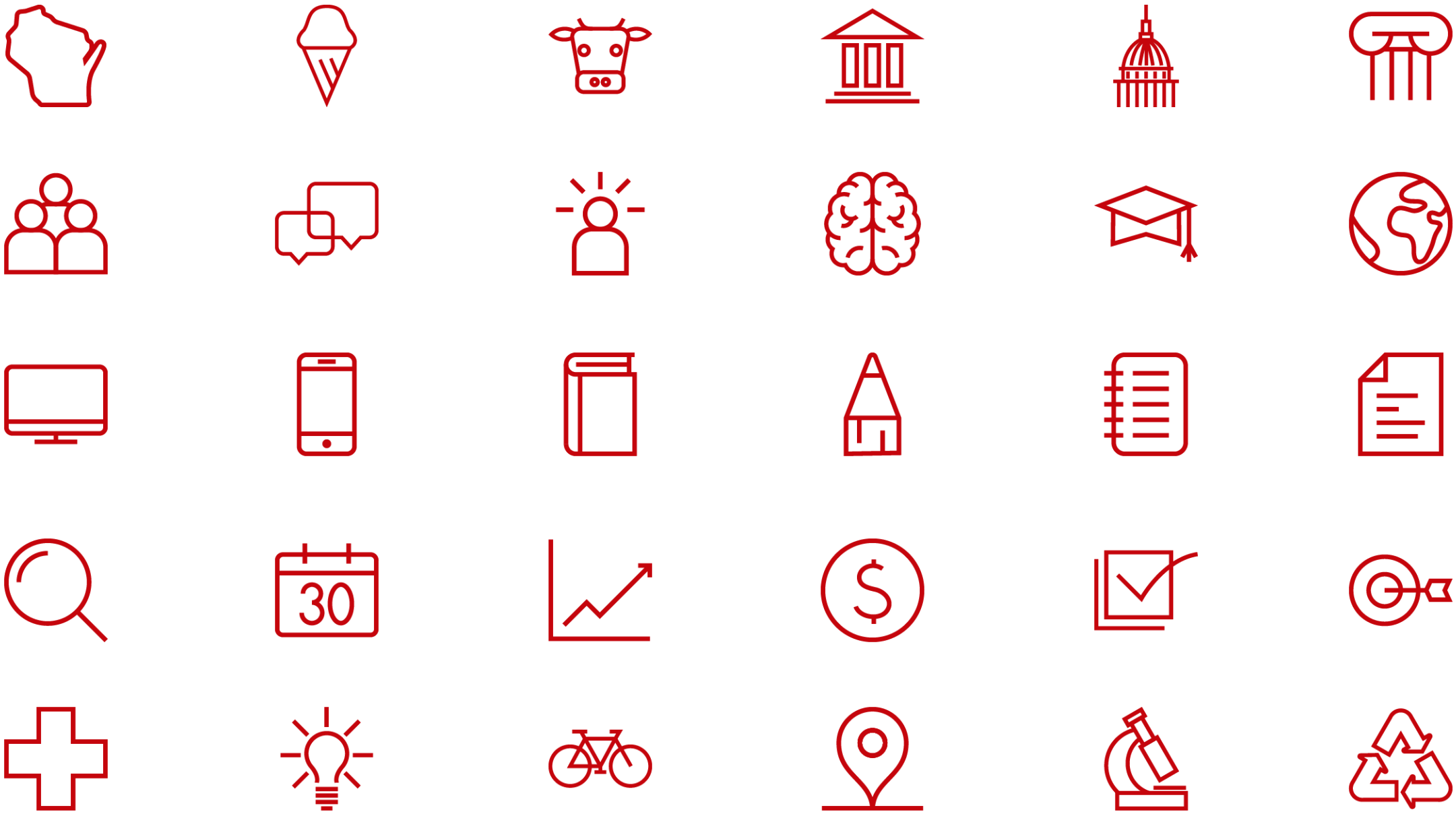
For big instances (> 5), DOCA Graph is much faster than Pthread and sequential code.

Because DOCA Graph builds the graph only once and submits all the instances for execution. Pthread synchronizes at every instance.

Conclusion

- We have presented the **motivation** behind DOCA Graph
- We have presented the DOCA Graph **programming model**
- We have presented an **application** using DOCA Graph
- We have presented the **performance** of DOCA Graph

Note: Only use red icons on white or light gray backgrounds




WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON