Order/Degree問題のための重みなしグラフにおける全点対間最短経路アルゴリズムの並列化

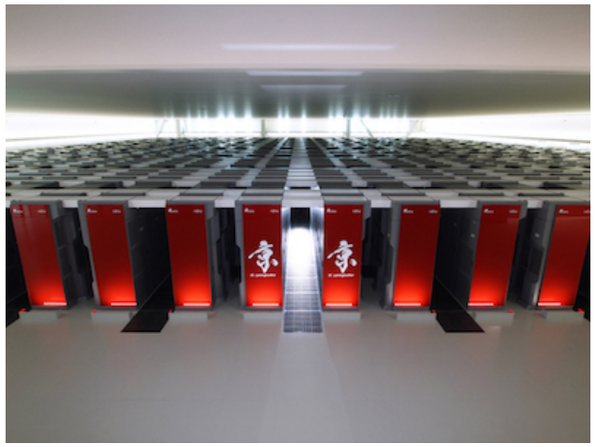
中尾昌広, 村井均, 佐藤三久

(理化学研究所 計算科学研究センター)

第171回HPC研究会@国立情報学研究所, 2019年9月20日

Background

- The network topology of a large-scale parallel computer system affects the overall performance
  - Supercomputer
  - Data center



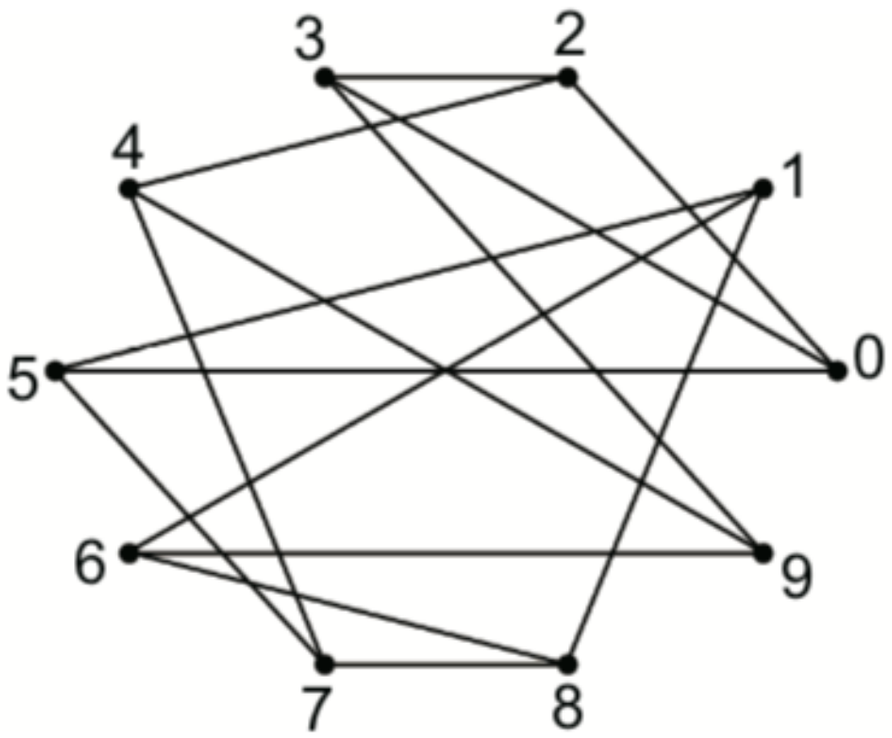
- It is important to design the network topology so that the **diameter** and **average distance** of the number of hops between calculation nodes are small

Designing such a network topology can be defined as an Order/Degree problem in graph theory

What's Order/Degree problem ?

- By considering the calculation node as “vertex” and the network wiring as “edge”, the network topology is represented as a graph
- The Order/Degree problem is to find the graph with the smallest diameter and average distance from a set of unweighted graphs with the given number of vertices (*n*) and degree (*d*)

(n, d) = (10, 3)



	0	1	2	3	4	5	6	7	8	9
0		2	1	1	2	1	3	2	3	2
1			3	3	3	1	1	2	1	2
2				1	1	2	3	2	3	2
3					2	2	2	3	3	1
4						2	2	1	2	1
5							2	1	2	3
6								2	1	1
7									1	2
8										2
9										

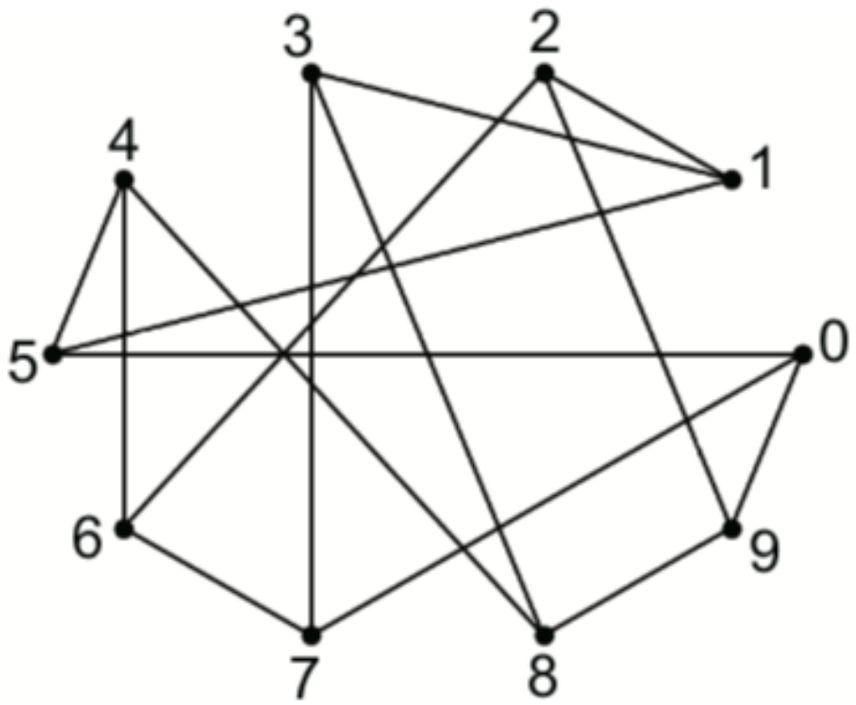
Dia. = 3
Ave. Dis. = 85/45
= 1.89..

全点対間最短経路（APSP）← All-pairs-shortest-path

What's Order/Degree problem ?

- By considering the calculation node as “vertex” and the network wiring as “edge”, the network topology is represented as a graph
- The Order/Degree problem is to find the graph with the smallest diameter and average distance from a set of unweighted graphs with the given number of vertices (*n*) and degree (*d*)

(n, d) = (10, 3)



	0	1	2	3	4	5	6	7	8	9
0		2	2	2	2	1	2	1	2	1
1			1	1	2	1	2	2	2	2
2				2	2	2	1	2	2	1
3					2	2	2	1	1	2
4						1	1	2	1	2
5							2	2	2	2
6								1	2	2
7									2	2
8										1
9										

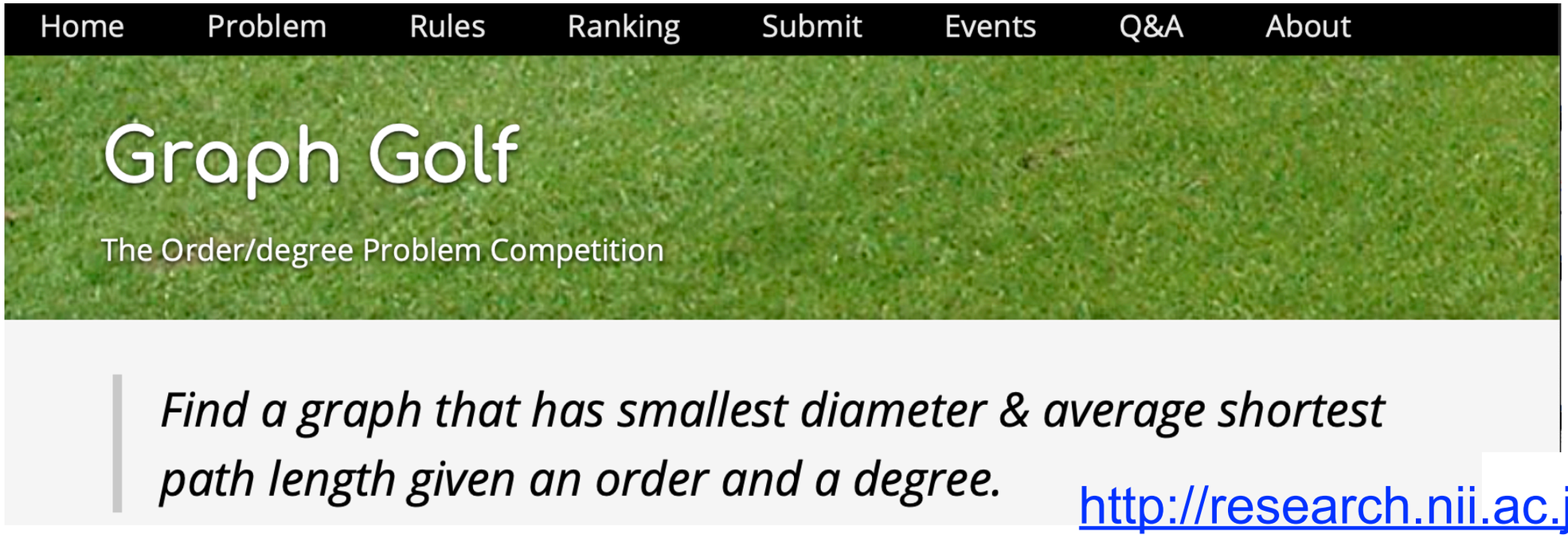
Dia. = 2
Ave. Dis. = 75/45
= 1.67..

全点対間最短経路（APSP）← All-pairs-shortest-path

Graph Golf

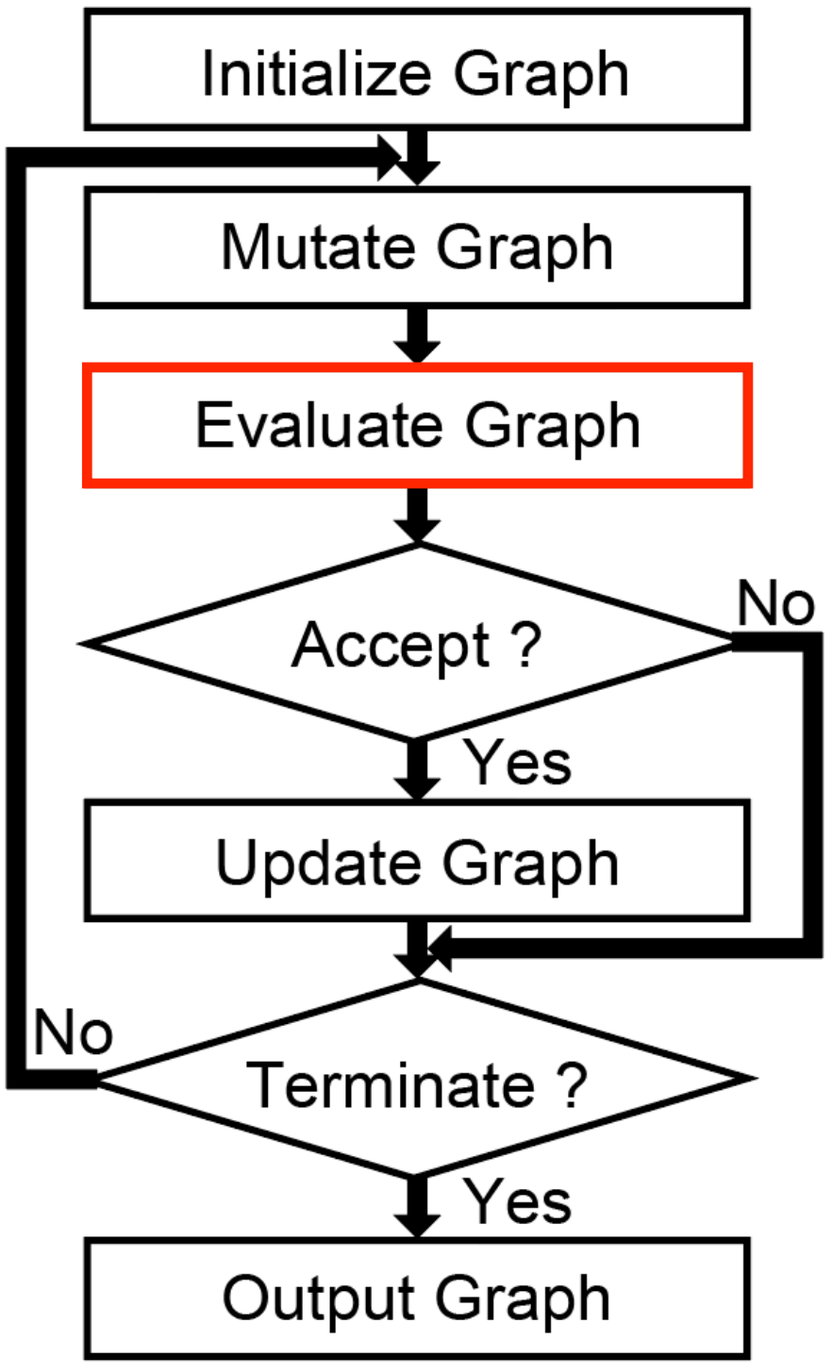
- International competition for the Order/Degree problem
  - Held by the National Institute of Informatics since 2015
  - Provides problems with combinations of *n* and *d* every year
  - The problems in 2019 are (n, d) = (50, 4), (512, 4), (512, 6), (1Ki, 4), (1726, 30), (4855, 15), (9344, 6), (64Ki, 6), (100K, 8), (100K, 16), (1M, 32)

K = 1,000, Ki = 1,024, M = 1,000,000



http://research.nii.ac.jp/graphgolf/

How to solve Order/Degree problem ?



- Metaheuristic algorithms such as Simulated Annealing are often used
- To execute the algorithms, it is necessary to calculate APSP many times
- Moreover, the computational cost of the APSP algorithm is very high !!

e.g. For a problem (n, d) = (1M, 32), the time required for one APSP is about 37 hours by the methods based on BFS on Intel Gold 6126

Objective and a part of results

- Our previous research provides an APSP algorithm based on Breadth-First Search (BFS-APSP) [2019nakao]
  - BFS-APSP is parallelized by OpenMP + MPI

- This research introduces another APSP algorithm based on adjacency matrix (ADJ-APSP) [2017mori], and compares BFS-APSP
  - ADJ-APSP is parallelized by MPI + OpenMP for multi-core cluster
  - ADJ-APSP is parallelized by MPI + CUDA for GPU cluster

BFS-APSP (about 37 hours) → ADJ-APSP (3,880 sec.) →
ADJ-APSP by OpenMP+MPI on 64 CPUs x 12 Cores (6.77 sec.) →
ADJ-APSP by CUDA+MPI on 128 GPUs (0.28 sec.)

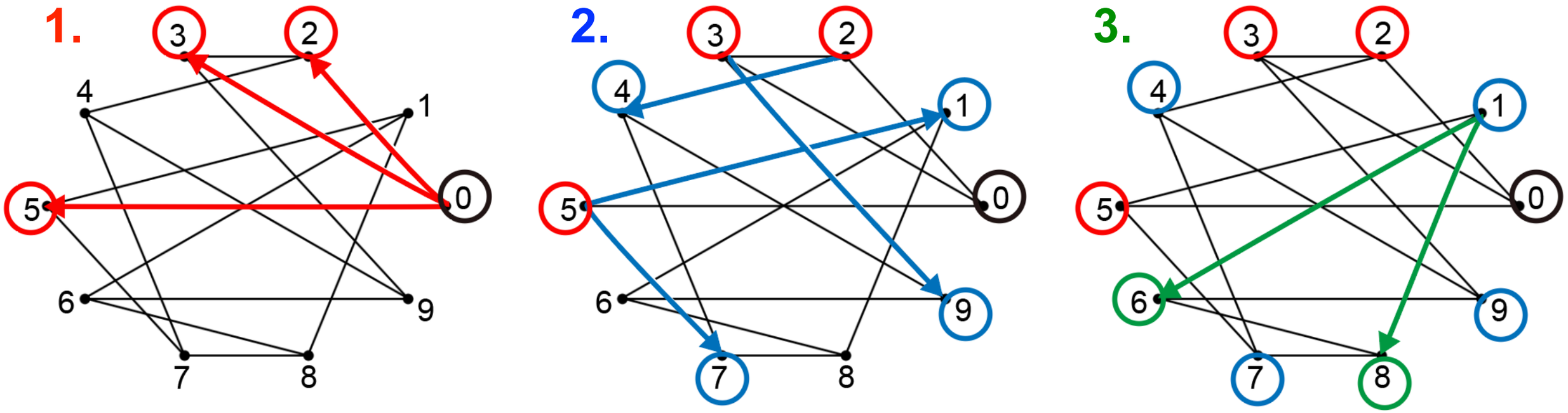
You can download programs from
https://github.com/mnakao/APSP/

Agenda

- Background
- BFS-APSP
- ADJ-APSP
- Performance
- Summary

Serial BFS-APSP

- BFS can be used to find the distances from one vertex to others
  - APSP can be obtained by BFS for all vertices
  - Top-down approach is used
- The computational complexity of applying BFS to one vertex is proportional to the number of edges; O(nd).
  - When it is repeated n times, the computational complexity of BFS-APSP is O(n^2d)



Parallel BFS-APSP

- Multiple BFSs are performed simultaneously using MPI, and one BFS is divided into threads using OpenMP
- For MPI,
  - Starting points are assigned to each MPI process evenly
  - Thus, the maximum number of processes is *n*
  - Communication time is small because only the information (diameter and average distance) of each process is collected at the end of the program
- For OpenMP,
  - Each OpenMP thread searches not-visited vertices
  - OpenMP requires exclusive control to update list of the not-visited vertices

Comparison with Graph500

- Graphs in Graph500
  - Kronecker graph where vertices with a large degree and vertices with a low degree are mixed
  - Like social networks
- Graphs in Order/Degree problem
  - Regular graph（正則グラフ）where degree ***d*** is constant
  - Like industrial products

From these conditions, we have confirmed that the top-down approach performs better than the hybrid approach[Beamer 2012] used in Graph500

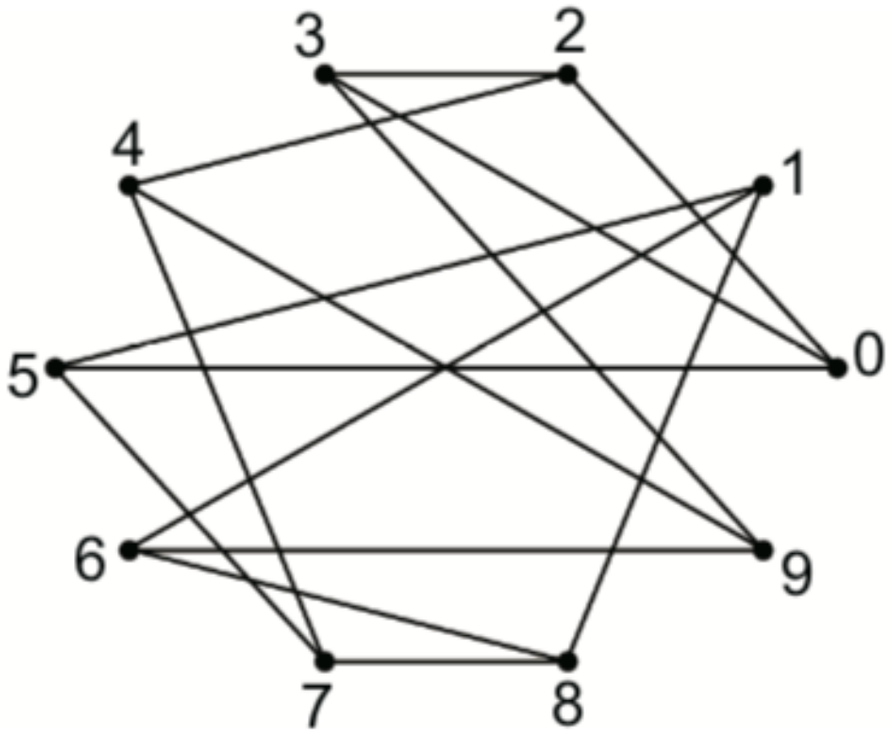
Agenda

- Background
- BFS-APSP
- ADJ-APSP
- Performance
- Summary

Serial ADJ-APSP(1/3)

- Let A be an adjacency matrix of a graph
- If the value of an element a_{i, j} in A^k is 1, it means that the vertex i can reach the vertex j within k hops

(n, d) = (10, 3)



adjacency list

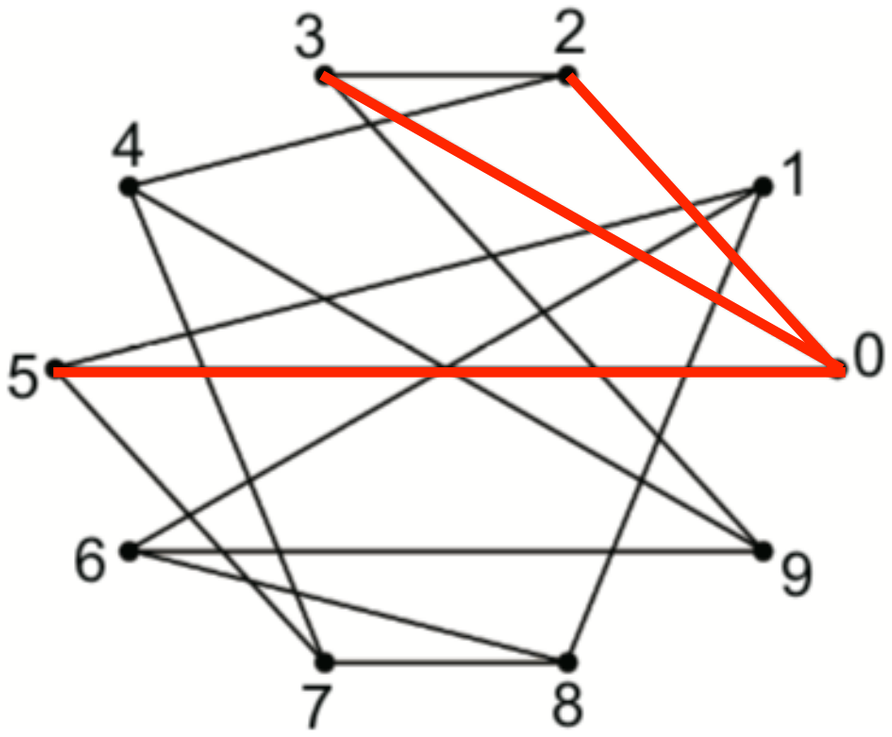
A	adjlst	A^1
0 0 0 0 0 0 0 0 0 1	2 3 5	0 0 0 0 **1** 0 **1** **1** 0 1
0 0 0 0 0 0 0 0 1 0	5 6 8	0 **1** 0 **1** **1** 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0	0 3 4	0 0 0 0 0 **1** **1** 1 0 **1**
0 0 0 0 0 0 1 0 0 0	0 2 9	**1** 0 0 0 0 0 1 **1** 0 **1**
0 0 0 0 0 1 0 0 0 0	2 7 9	**1** 0 **1** 0 0 1 0 **1** 0 0
0 0 0 0 1 0 0 0 0 0	0 1 7	0 0 **1** 0 1 0 0 0 **1** **1**
0 0 0 1 0 0 0 0 0 0	1 8 9	**1** **1** 0 1 0 0 0 0 **1** 0
0 0 1 0 0 0 0 0 0 0	4 5 8	0 **1** 1 0 **1** **1** 0 0 0 0
0 1 0 0 0 0 0 0 0 0	1 6 7	0 1 **1** **1** 0 0 0 0 **1** 0
1 0 0 0 0 0 0 0 0 0	3 4 6	1 0 0 **1** 0 **1** **1** 0 0 0

Serial ADJ-APSP(1/3)

- Let A be an adjacency matrix of a graph
- If the value of an element a_{i, j} in A^k is 1, it means that the vertex i can reach the vertex j within k hops

```
for(int i=0;i<n;i++)
  A[0][i] |= A[2][i] | A[3][i] | A[5][i];
```

(n, d) = (10, 3)



A	adjlst	A^1
0 0 0 0 0 0 0 0 0 1	2 3 5	0 0 0 0 **1** 0 **1** **1** 0 1
0 0 0 0 0 0 0 0 1 0	5 6 8	0 **1** 0 **1** **1** 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0	0 3 4	0 0 0 0 0 **1** **1** 1 0 **1**
0 0 0 0 0 0 1 0 0 0	0 2 9	**1** 0 0 0 0 0 1 **1** 0 **1**
0 0 0 0 0 1 0 0 0 0	2 7 9	**1** 0 **1** 0 0 1 0 **1** 0 0
0 0 0 0 1 0 0 0 0 0	0 1 7	0 0 **1** 0 1 0 0 0 **1** **1**
0 0 0 1 0 0 0 0 0 0	1 8 9	**1** **1** 0 1 0 0 0 0 **1** 0
0 0 1 0 0 0 0 0 0 0	4 5 8	0 **1** 1 0 **1** **1** 0 0 0 0
0 1 0 0 0 0 0 0 0 0	1 6 7	0 1 **1** **1** 0 0 0 0 **1** 0
1 0 0 0 0 0 0 0 0 0	3 4 6	1 0 0 **1** 0 **1** **1** 0 0 0

Serial ADJ-APSP(2/3)

A	adjlst	A^1	A^2	A^3
0 0 0 0 0 0 0 0 0 1	2 3 5	0 0 0 0 ***1*** 0 ***1*** ***1*** 0 1	***1*** 0 ***1*** 0 1 ***1*** 1 1 ***1*** 1	1 ***1*** 1 ***1*** 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 0	5 6 8	0 ***1*** 0 ***1*** ***1*** 0 0 0 1 0	***1*** 1 ***1*** 1 1 0 0 0 1 ***1***	1 1 1 1 1 ***1*** ***1*** ***1*** 1 1
0 0 0 0 0 0 0 1 0 0	0 3 4	0 0 0 0 0 ***1*** ***1*** 1 0 ***1***	***1*** 0 ***1*** 0 ***1*** 1 1 1 0 1	1 ***1*** 1 ***1*** 1 1 1 1 ***1*** 1
0 0 0 0 0 0 1 0 0 0	0 2 9	***1*** 0 0 0 0 0 1 ***1*** 0 ***1***	1 0 0 ***1*** ***1*** ***1*** 1 1 0 1	1 ***1*** ***1*** 1 1 1 1 1 ***1*** 1
0 0 0 0 0 1 0 0 0 0	2 7 9	***1*** 0 ***1*** 0 0 1 0 ***1*** 0 0	1 ***1*** 1 ***1*** ***1*** 1 ***1*** 1 0 ***1***	1 1 1 1 1 1 1 1 ***1*** 1
0 0 0 0 1 0 0 0 0 0	0 1 7	0 0 ***1*** 0 1 0 0 0 ***1*** ***1***	0 ***1*** 1 ***1*** 1 ***1*** ***1*** 1 1	***1*** 1 1 1 1 1 1 1 1 1
0 0 0 1 0 0 0 0 0 0	1 8 9	***1*** ***1*** 0 1 0 0 0 0 ***1*** 0	1 1 ***1*** 1 ***1*** ***1*** 0 1 0	1 1 1 1 1 1 1 ***1*** 1 ***1***
0 0 1 0 0 0 0 0 0 0	4 5 8	0 ***1*** 1 0 ***1*** ***1*** 0 0 0 0	***1*** 1 1 ***1*** 1 1 0 ***1*** ***1***	1 1 1 1 1 1 ***1*** 1 1 1
0 1 0 0 0 0 0 0 0 0	1 6 7	0 1 ***1*** ***1*** 0 0 0 0 ***1*** 0	***1*** 1 1 1 ***1*** ***1*** 0 0 1 0	1 1 1 1 1 1 ***1*** ***1*** 1 ***1***
1 0 0 0 0 0 0 0 0 0	3 4 6	1 0 0 ***1*** 0 ***1*** ***1*** 0 0 0	1 ***1*** ***1*** 1 0 1 1 ***1*** ***1*** ***1***	1 1 1 1 ***1*** 1 1 1 1 1

- As k is increased in increments of 1, the value of k is the diameter when all elements are 1
- Every time k is increased from 1 to the diameter, the average distance can be obtained by summing all the elements whose value for element a_{i, j} in A^k is 0 divided by number of elements

Serial ADJ-APSP(3/3)

```
1  function SERIAL_ADJ_APSP(vertices, nodes)
2     diameter ← 1
3     distance ← nodes*(nodes−1)
4     elements ← ⌈nodes/E⌉
5     A, B ← INITIALIZE(nodes, elements)
6     for k=1 ... nodes−1
7        for i=1 ... nodes
8           for n ∈ neighbors(i, vertices)
9              for j=1 ... elements
10                B[i][j] ← B[i][j] | A[n][j]
11
12       num ← 0
13       for i=1 ... nodes
14          for j=1 ... elements
15             num ← num+POPCNT(B[i][j])
16
17       if(num = nodes*nodes) break
18
19       SWAP(A, B)
20       diameter++
21       distance ← distance+(nodes*nodes−num)
22    average_distance ← distance/((nodes−1)*nodes)
23    return diameter, average_distance
```

← logical sum operation
(the most time-consuming part)

←__builtin_popcountll() or _mm_popcnt_u64()

Parallel ADJ-APSP

A	adjlst	A^1
0 0 0 0 0 0 0 0 0 1	2 3 5	0 0 0 0 ***1*** 0 ***1*** ***1*** 0 1
0 0 0 0 0 0 0 0 1 0	5 6 8	0 ***1*** 0 ***1*** ***1*** 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0	0 3 4	0 0 0 0 0 ***1*** ***1*** 1 0 ***1***
0 0 0 0 0 0 1 0 0 0	0 2 9	***1*** 0 0 0 0 0 1 ***1*** 0 ***1***
0 0 0 0 0 1 0 0 0 0	2 7 9	***1*** 0 ***1*** 0 0 1 0 ***1*** 0 0
0 0 0 0 1 0 0 0 0 0	0 1 7	0 0 ***1*** 0 1 0 0 0 ***1*** ***1***
0 0 0 1 0 0 0 0 0 0	1 8 9	***1*** ***1*** 0 1 0 0 0 0 ***1*** 0
0 0 1 0 0 0 0 0 0 0	4 5 8	0 ***1*** 1 0 ***1*** ***1*** 0 0 0 0
0 1 0 0 0 0 0 0 0 0	1 6 7	0 1 ***1*** ***1*** 0 0 0 0 ***1*** 0
1 0 0 0 0 0 0 0 0 0	3 4 6	1 0 0 ***1*** 0 ***1*** ***1*** 0 0 0

- "A" can be calculated independently by MPI
- The maximum number of processes is (*n/E*), E is number of bit in one elements (we use uint64_t, E = 64)

Divide "A" vertically

A_1	adjlst	A_1^1	A_0	adjlst	A_0^1
0 0 0 0 0	2 3 5	0 0 0 0 ***1***	0 0 0 0 1	2 3 5	0 ***1*** ***1*** 0 1
0 0 0 0 0	5 6 8	0 ***1*** 0 ***1*** ***1***	0 0 0 1 0	5 6 8	0 0 0 1 0
0 0 0 0 0	0 3 4	0 0 0 0 0	0 0 1 0 0	0 3 4	***1*** ***1*** 1 0 ***1***
0 0 0 0 0	0 2 9	***1*** 0 0 0 0	0 1 0 0 0	0 2 9	0 1 ***1*** 0 ***1***
0 0 0 0 0	2 7 9	***1*** 0 ***1*** 0 0	1 0 0 0 0	2 7 9	1 0 ***1*** 0 0
0 0 0 0 1	0 1 7	0 0 ***1*** 0 1	0 0 0 0 0	0 1 7	0 0 0 ***1*** ***1***
0 0 0 1 0	1 8 9	***1*** ***1*** 0 1 0	0 0 0 0 0	1 8 9	0 0 0 ***1*** 0
0 0 1 0 0	4 5 8	0 ***1*** 1 0 ***1***	0 0 0 0 0	4 5 8	***1*** 0 0 0 0
0 1 0 0 0	1 6 7	0 1 ***1*** ***1*** 0	0 0 0 0 0	1 6 7	0 0 0 ***1*** 0
1 0 0 0 0	3 4 6	1 0 0 ***1*** 0	0 0 0 0 0	3 4 6	***1*** ***1*** 0 0 0

Parallel ADJ-APSP

```
1  function PARALLEL_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← 0
4    elements ← ⌈nodes/E⌉
5    chunk ← ⌈elements/procs⌉
6    parsize ← ⌈elements/chunk⌉
7    for c=1 ... parsize on each process
8      A, B ← INITIALIZE(nodes, chunk)
9      for k=1 .. nodes−1
10       for i=1 ... nodes omp parallel
11         for n ∈ neighbors(i, vertices)
12           for j=1 ... chunk
13             B[i][j] ← B[i][j] | A[n][j]
14
15       num ← 0
16       for i=1 ... nodes omp parallel reduction(+:num)
17         for j=1 ... chunk
18           num ← num+POPCNT(B[i][j])
19
20       if(num = nodes*chunk*E) break
21
22       SWAP(A, B)
23       distance ← distance+(nodes*chunk*E−num)
24     diameter ← MAX(diameter, k+1)
25   diameter ← ALLREDUCE(diameter, MAX)
26   average_distance ← ALLREDUCE(distance, SUM)
27   average_distance ← average_distance/((nodes−1)*nodes)+1
28   return diameter, average_distance
```

OpenMP

OpenMP

Same as BFS-APSP, communication time is very small

Parallel ADJ-APSP for GPU

```
1  function PARALLEL_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← 0
4    elements ← ⌈nodes/E⌉
5    chunk ← ⌈elements/procs⌉
6    parsize ← ⌈elements/chunk⌉
7    for c=1 ... parsize on each process
8      A, B ← INITIALIZE(nodes, chunk)
9      for k=1 .. nodes−1
10       for i=1 ... nodes omp parallel
11         for n ∈ neighbors(i, vertices)
12           for j=1 ... chunk
13             B[i][j] ← B[i][j] | A[n][j]
14
15       num ← 0
16       for i=1 ... nodes omp parallel reduction(+:num)
17         for j=1 ... chunk
18           num ← num+POPCNT(B[i][j])
19
20       if(num = nodes*chunk*E) break
21
22       SWAP(A, B)
23       distance ← distance+(nodes*chunk*E−num)
24     diameter ← MAX(diameter, k+1)
25   diameter ← ALLREDUCE(diameter, MAX)
26   average_distance ← ALLREDUCE(distance, SUM)
27   average_distance ← average_distance/((nodes−1)*nodes)+1
28   return diameter, average_distance
```

in CUDA

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;

while (tid < nodes*elements) {
  int i = tid / elements;
  int k = tid % elements;
  uint64_t tmp = B[i][k];
  for(int j=0;j<num_degrees[i];j++){
    int n = neighbors[i][j];
    tmp |= A[n][k];
  }
  B[i][k] = tmp;
  tid += blockDim.x * gridDim.x;
}
```

__popcnt() in CUDA Math API

Comparison between BFS-APSP and ADJ-APSP

	BFS-APSP	ADJ-APSP
Computational complexity	O(n^2d)	O(n^2dD/E)
Maximum number of MPI processes	n	n/E
OpenMP exclusive control	critical directive	(none)
For GPU	△	○
Communication	MPI_Allreduce() x 2 for scalar	

n: vertices
d: degree
D: diameter
E: bits in elements (64)

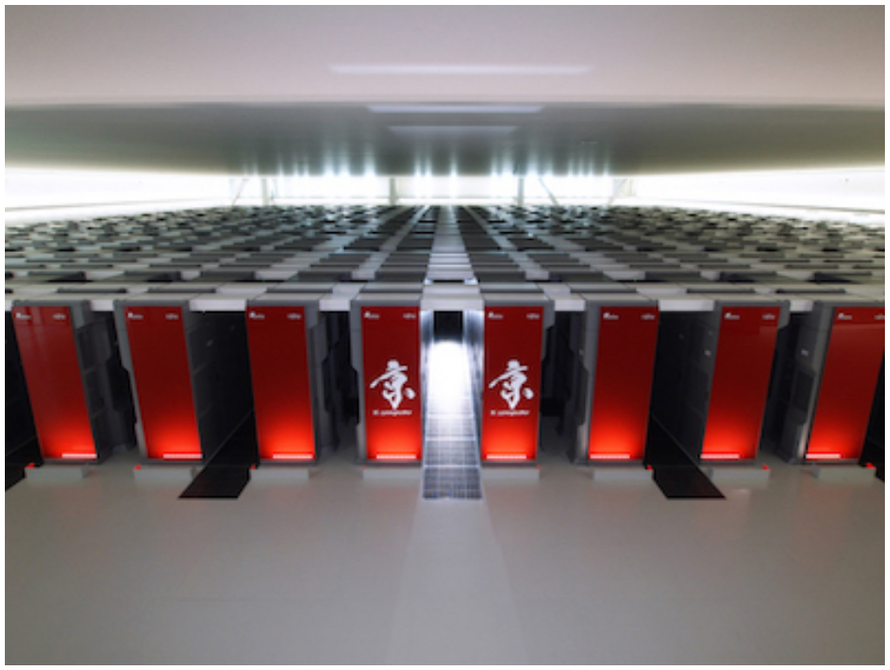
In general, the value of D of graphs in Order/Degree problem is small due to the small-world effect.

Agenda

- Background
- BFS-APSP
- ADJ-APSP
- Performance
- Summary

Experiment environment

The K computer in RIKEN R-CCS



CPU	SPARC64 VIIIfx (8Cores, 2.0GHz)
Memory	DDR3 (64GB/s, 16GB)
Network	Torus fusion six-dimensional mesh/torus network, 5GB/s × 10
Software	Fujitsu Compiler K-1.2.0-25

For OpenMP+MPI versions

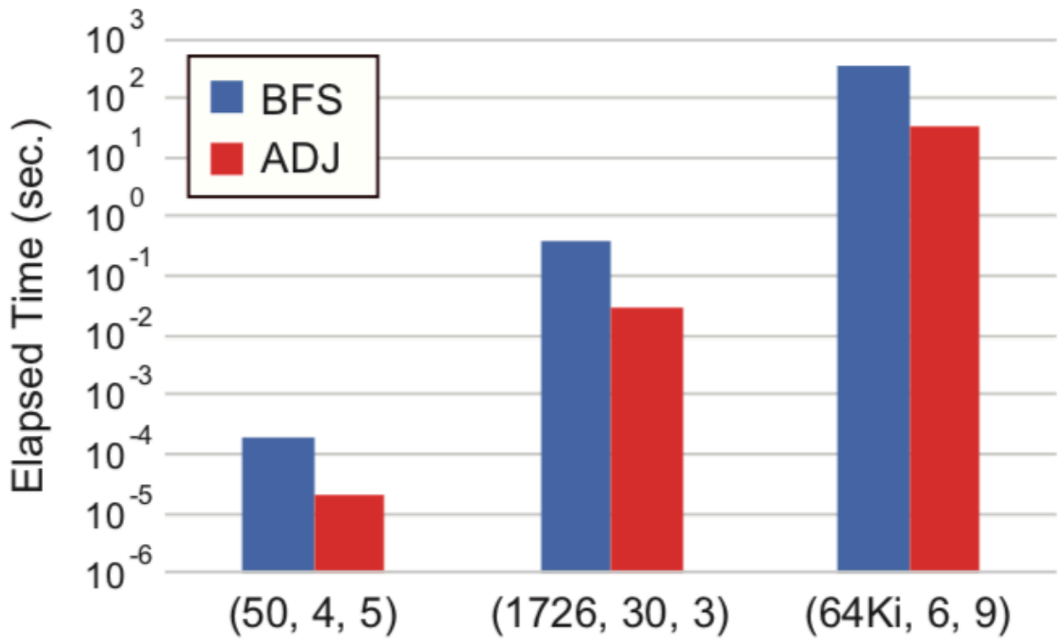
Cygnus in CCS, Univ. of Tsukuba

CPU	Intel Xeon Gold 6126 (12Cores, 2.6GHz) × 2
Memory	DDR4 (128GB/s × 2, 192GB)
GPU	NVIDIA Tesla V100 (900GB/s, 32GB) × 4
Network	InfiniBand HDR100 (12.5GB/s) × 4
Software	intel/19.0.3, mvapich/2.3.1, cuda/10.1

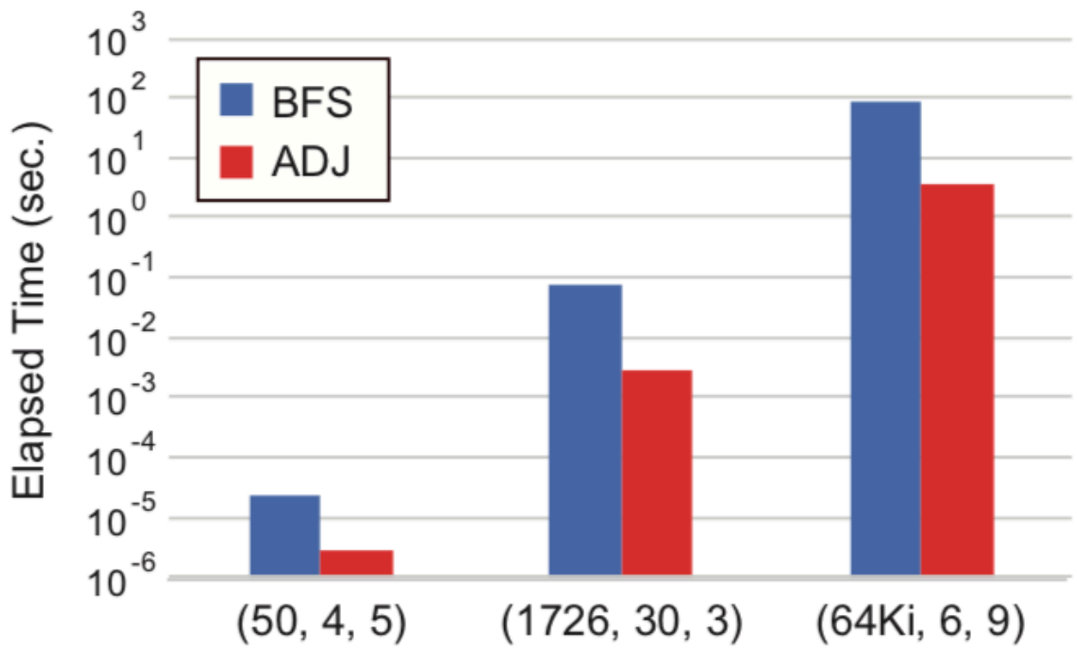
For OpenMP+MPI versions
For CUDA+MPI version

Serial algorithm

The K computer



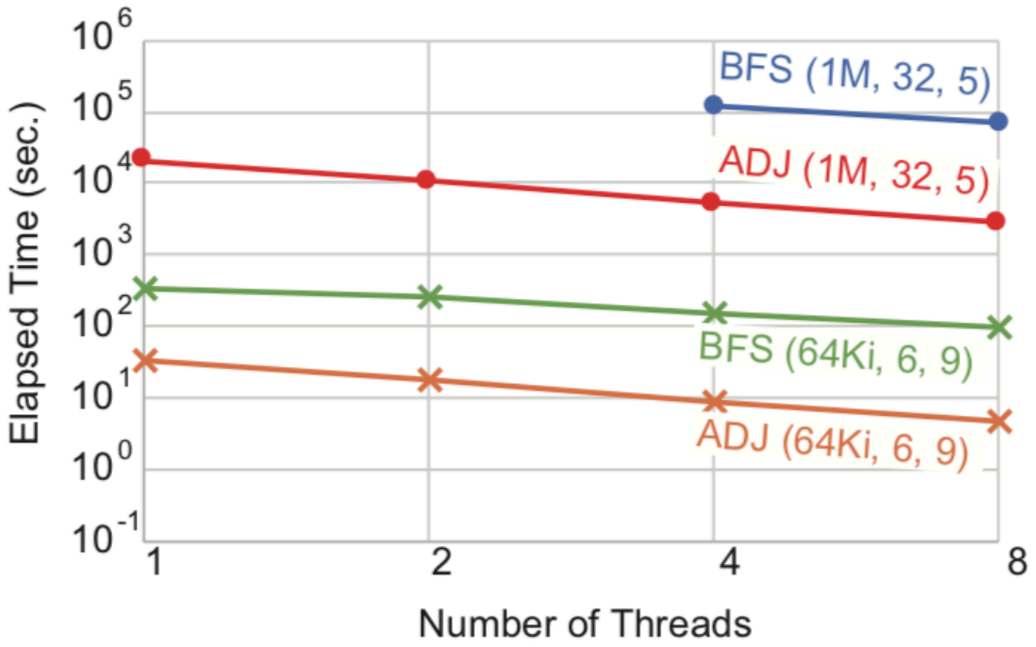
Cygnus system



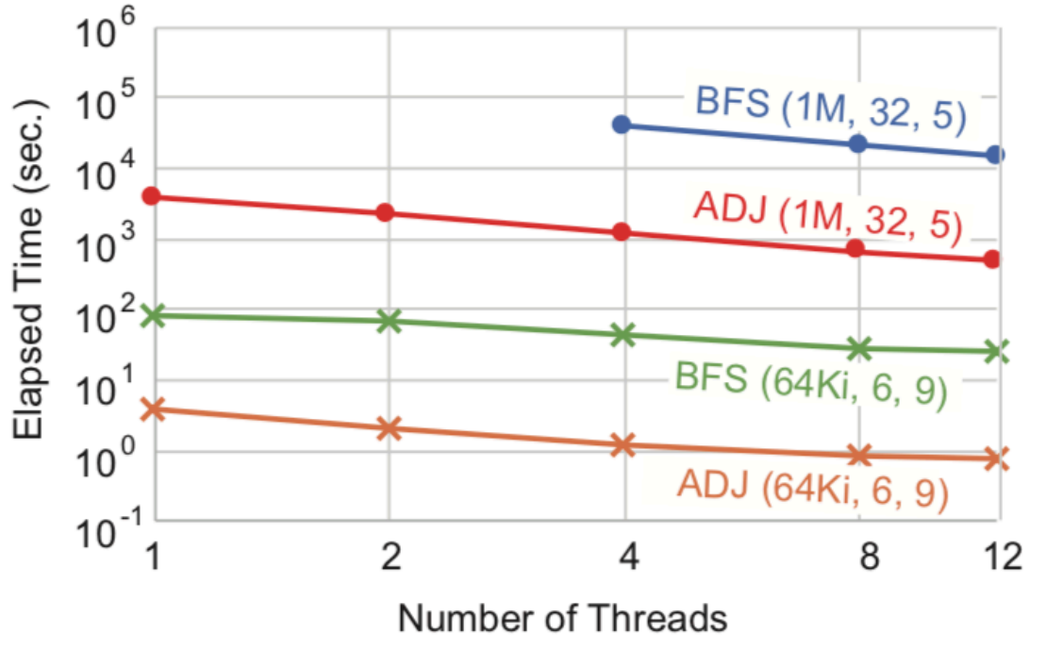
- (n, d, D) = (50, 4, 5), (1726, 30, 3), and (64Ki, 6, 9)
- ADJ-APSP is always faster than BFS-APSP
  - The computation time is 8.08 to 29.49 times faster

Parallel algorithm by OpenMP

The K computer



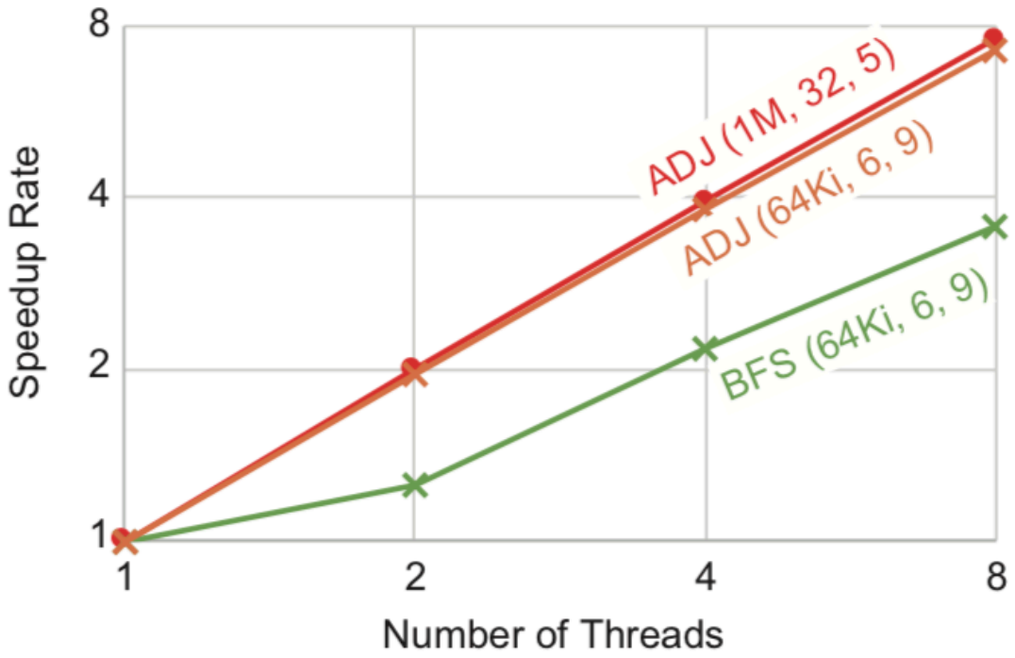
Cygnus system



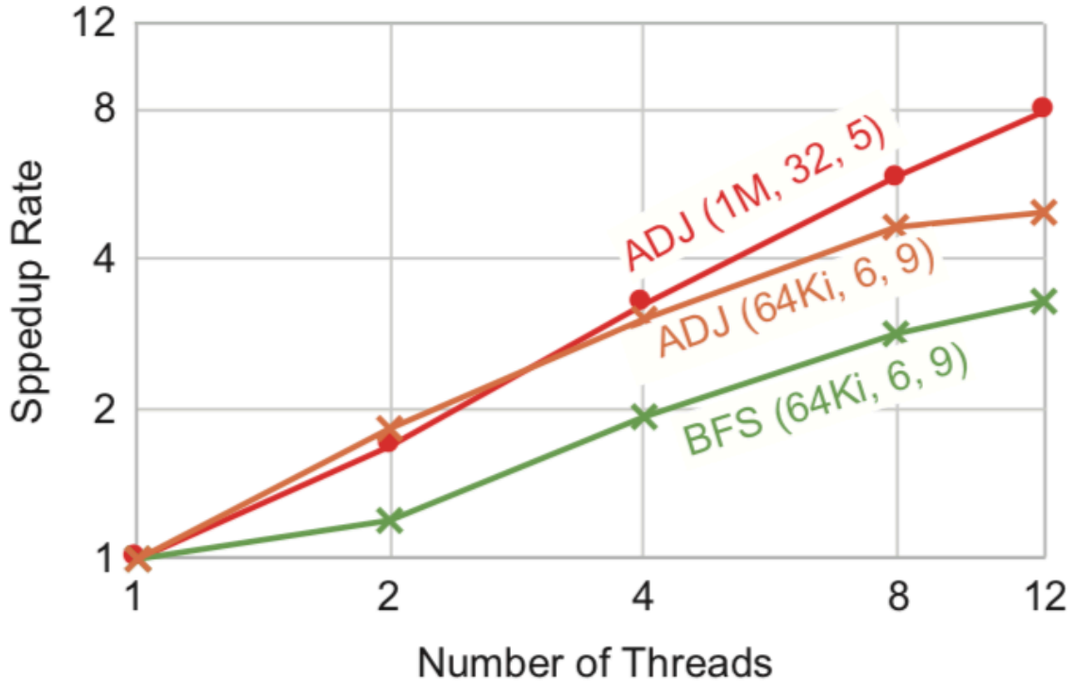
- (n, d, D) = (64Ki, 6, 9) and (1M, 32, 5)
- The number of processes is fixed at 1
- ADJ-APSP is always faster than BFS-APSP
  - 19.62 to 32.34 times faster at the maximum number of threads
  - (1M, 32, 5) on Cygnus
    - BFS-APSP(1core) : approx. 37hours → ADJ-APSP(1core) : 3,880sec.
    - ADJ-APSP(1core) : 3,880sec. → ADJ-APSP(12cores) : 475sec.

Parallel algorithm by OpenMP

The K computer



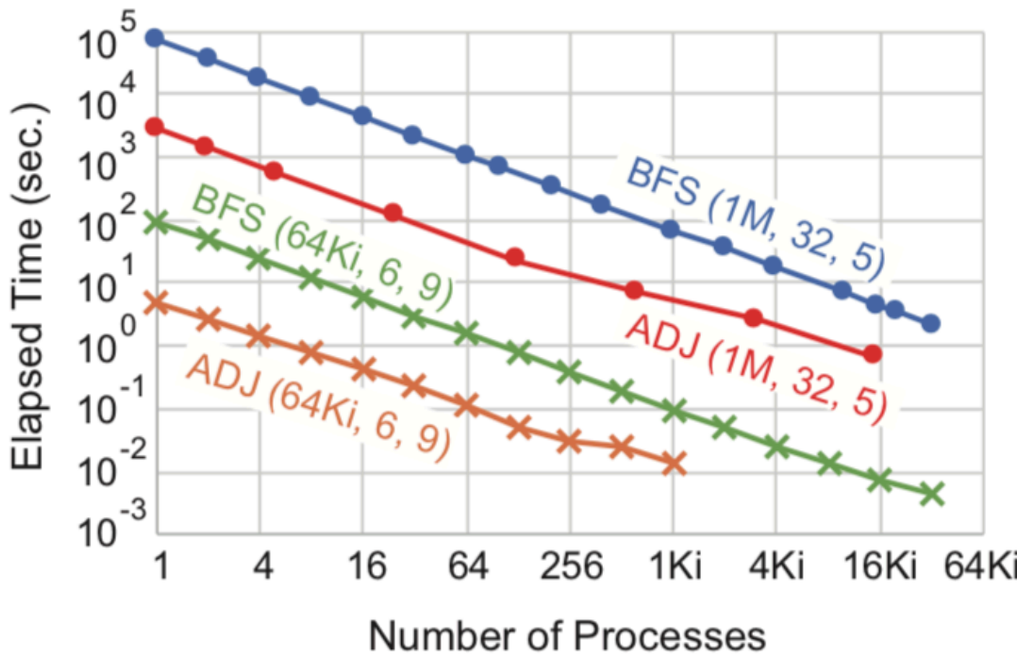
Cygnus system



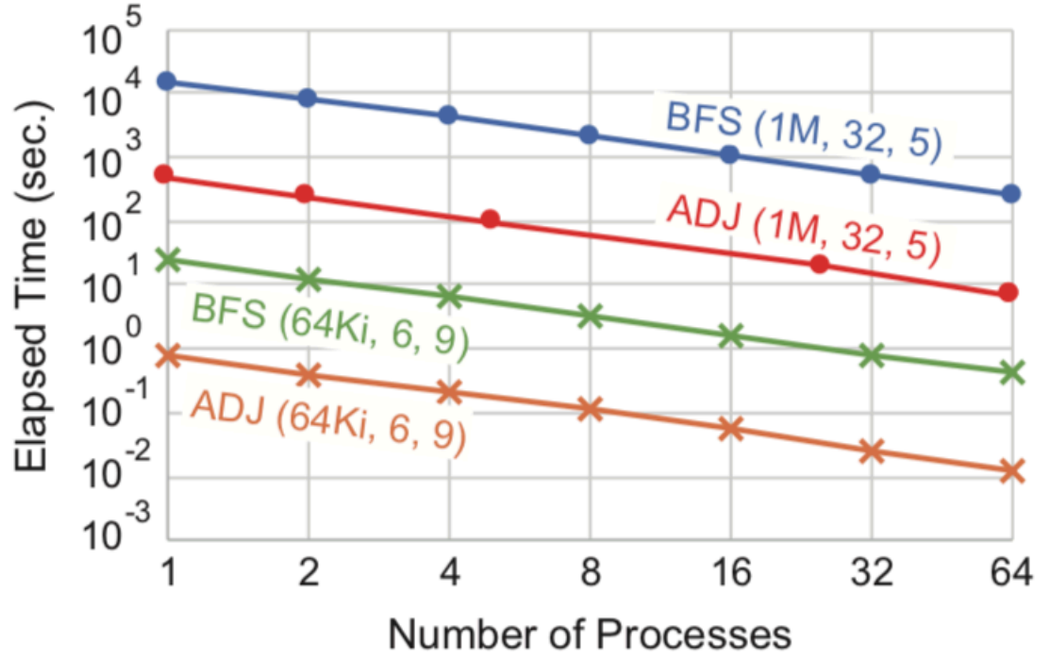
- Above graphs show the speed increase for previous graphs
- Parallelization efficiency of ADJ-APSP is higher than that of BFS-APSP
- This is because BFS-APSP requires exclusive control between threads, whereas ADJ-APSP does not perform such a control

Parallel algorithm by OpenMP+MPI

The K computer

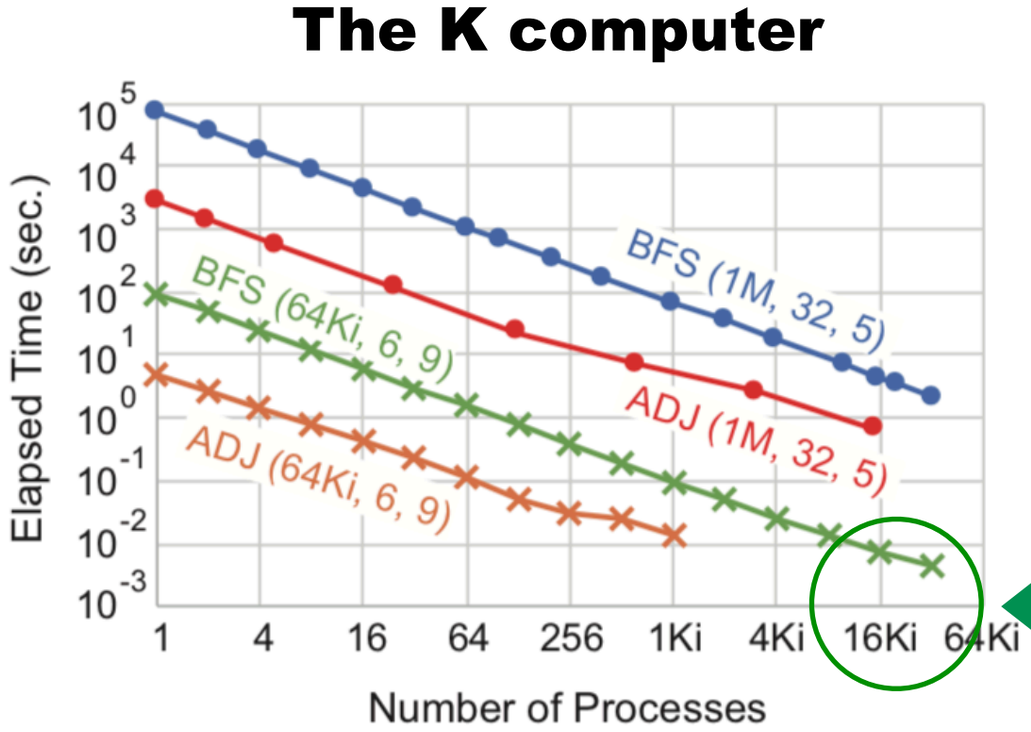


Cygnus system



- The number of threads is set to the maximum value
- The maximum number of processes in (64Ki, 6, 9) and (1M, 32, 5) is 65,536 and 1,000,000 for BFS-APSP and 1,024 and 15,625 for ADJ-APSP, respectively
- ADJ-APSP is faster than BFS-APSP for the same number of processes
- (1M, 32, 5) on cygnus
  - ADJ-APSP(1CPU) : 475sec. → ADJ-APSP(64CPU) : 6.77 sec.

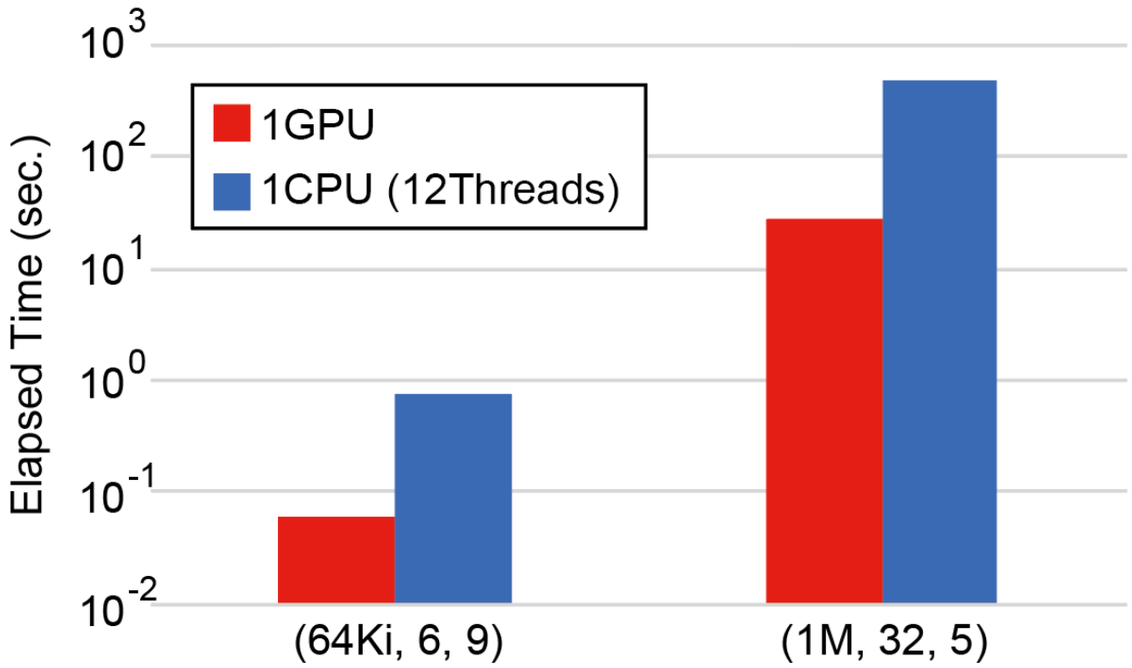
Parallel algorithm by OpenMP+MPI

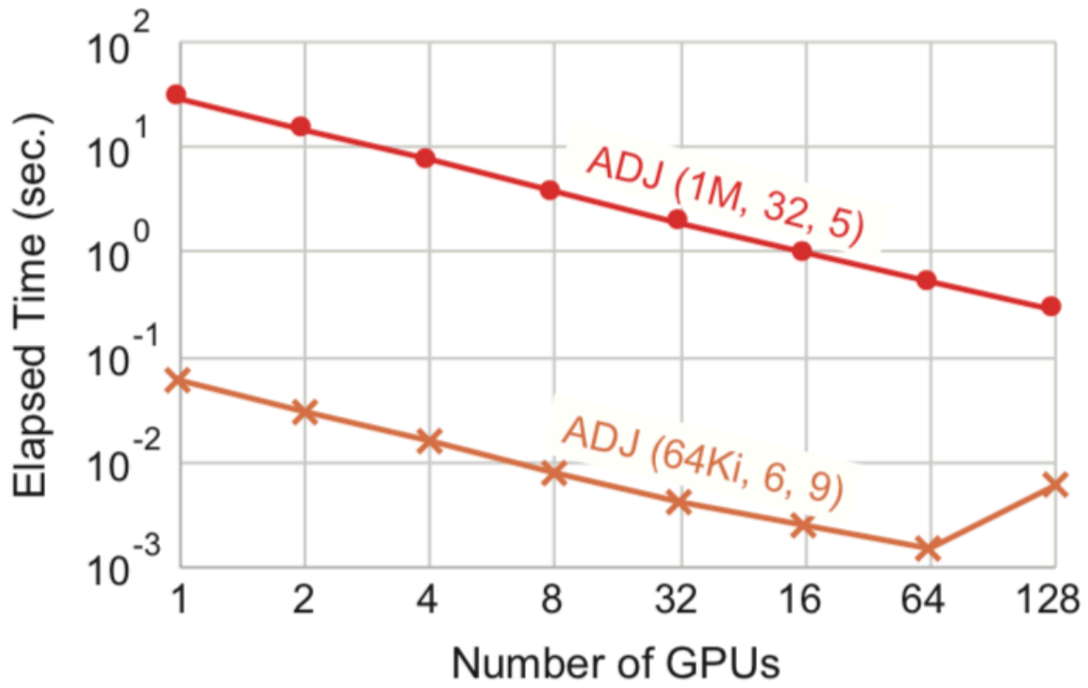


BFS-APSP may be faster than ADJ-APSP in a large number of processes

- The number of threads is set to the maximum value
- The maximum number of processes in (64Ki, 6, 9) and (1M, 32, 5) is 65,536 and 1,000,000 for BFS-APSP and 1,024 and 15,625 for ADJ-APSP, respectively
- ADJ-APSP is faster than BFS-APSP for the same number of processes
- (1M, 32, 5) on cygnus
  - ADJ-APSP(1CPU) : 475sec. → ADJ-APSP(64CPU) : 6.77 sec.

Parallel algorithm by CUDA+MPI

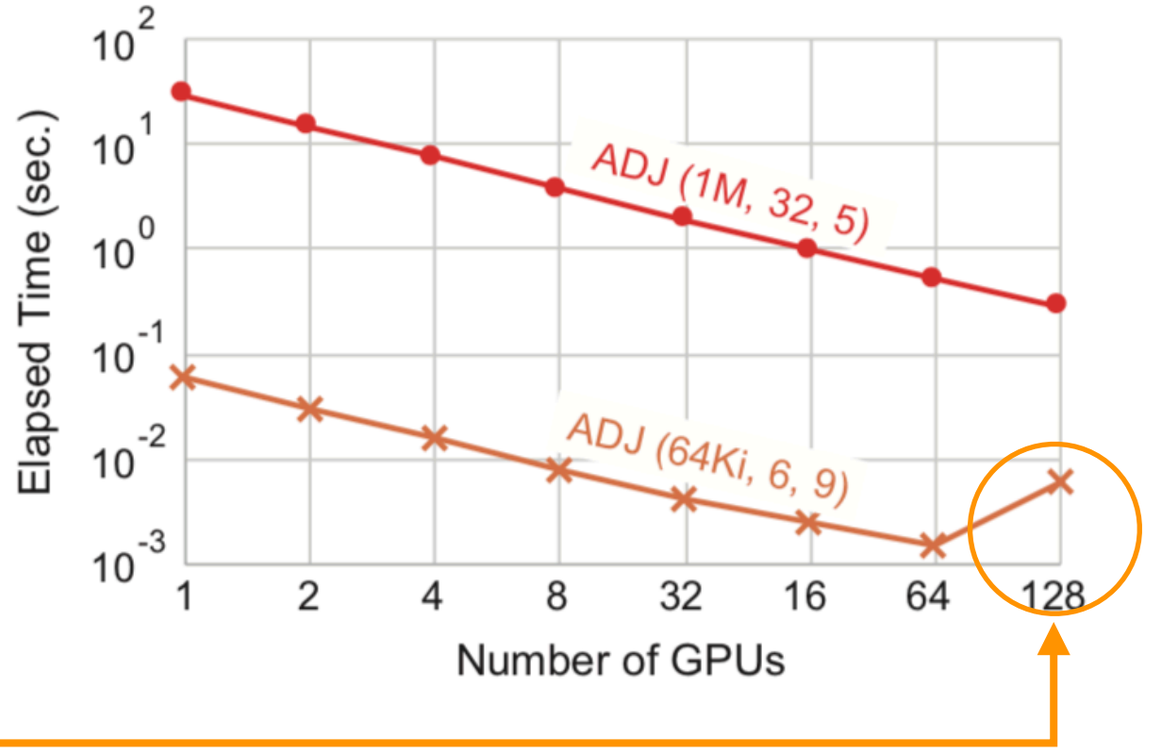




- (64Ki, 6, 9) : 0.77sec. (1CPU, 12Threads) → 0.06 sec. (1GPU) : x 12.6
- (1M, 32, 5) : 475 sec. (1CPU, 12Threads) → 28.7 sec. (1GPU) : x 16.5
- (1M, 32, 5) : 28.7sec.(1GPU) → 0.28 sec. (128GPUs)
  - Achieve 101.10-fold performance improvements

Parallel algorithm by CUDA+MPI

Since the number of elements of each column in the adjacency matrix is 8 (=65536/E/128 = 8), the condition where coalesce access occurs isn't met



- (64Ki, 6, 9) : 0.77sec. (1CPU, 12Threads) → 0.06 sec. (1GPU) : x 12.6
- (1M, 32, 5) : 475 sec. (1CPU, 12Threads) → 28.7 sec. (1GPU) : x 16.5
- (1M, 32, 5) : 28.7sec.(1GPU) → 0.28 sec. (128GPUs)
  - Achieve 101.10-fold performance improvements

Agenda

- Background
- BFS-APSP
- ADJ-APSP
- Performance
- Summary

Summary

- We parallelize BFS-APSP and ADJ-APSP using MPI+OpenMP for Order/Degree problem
- ADJ-APSP has a better performance in the serial algorithm and threaded algorithm than BFS-APSP on a single CPU
  - approx. 37hours (BFS, 1core) → 3,880sec. (ADJ, 1core) → 475sec. (ADJ, 12cores)
- However, because the maximum number of processes of BFS-APSP is larger than that of ADJ-APSP, the performance of BFS-APSP on multiple CPUs may be higher using MPI
- We achieved further speedup by parallelizing ADJ-APSP using GPUs
  - 28.7sec. (ADJ, 1GPU) → 0.28 sec.(ADJ, 128GPUs)