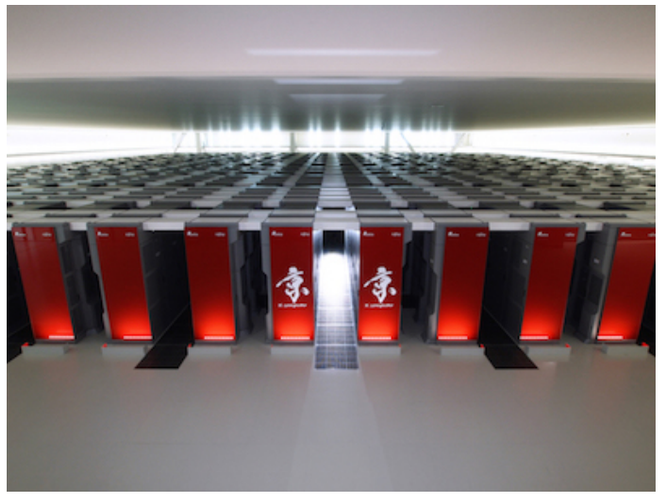
Parallelization of All-Pairs-Shortest-Path Algorithms in Unweighted Graph

Masahiro Nakao, Hitoshi Murai, Mitsuhisa Sato

(RIKEN Center for Computational Science)

Background (1/3)

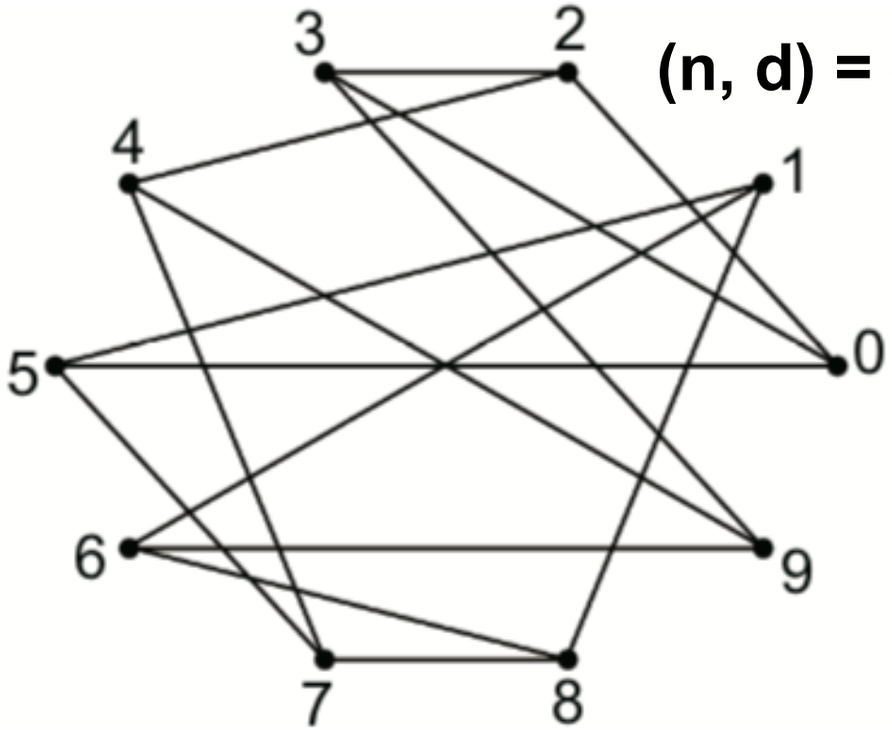
- Network topology of a large-scale parallel computer system affects the overall performance significantly
  - Supercomputer
  - Data center



- For a parallel application where the communication pattern is NOT known in advance (e.g. Big data), some researches consider that **a network topology with randomness is effective**
- The reason is that **diameter** and **average hops** of the random graph are smaller than those of a regular graph (e.g. k-ary n-cube) due to the small-world effect

Background (2/3)

- By describing the calculation node as “vertex” and the network wiring as “edge”, the network topology is represented as an unweighted graph with number of vertices (*n*) and degree (*d*)
- **Diameter** and **average hops** are important metrics of network topology, they are calculated using **All-Pairs-Shortest-Path (APSP)** algorithm

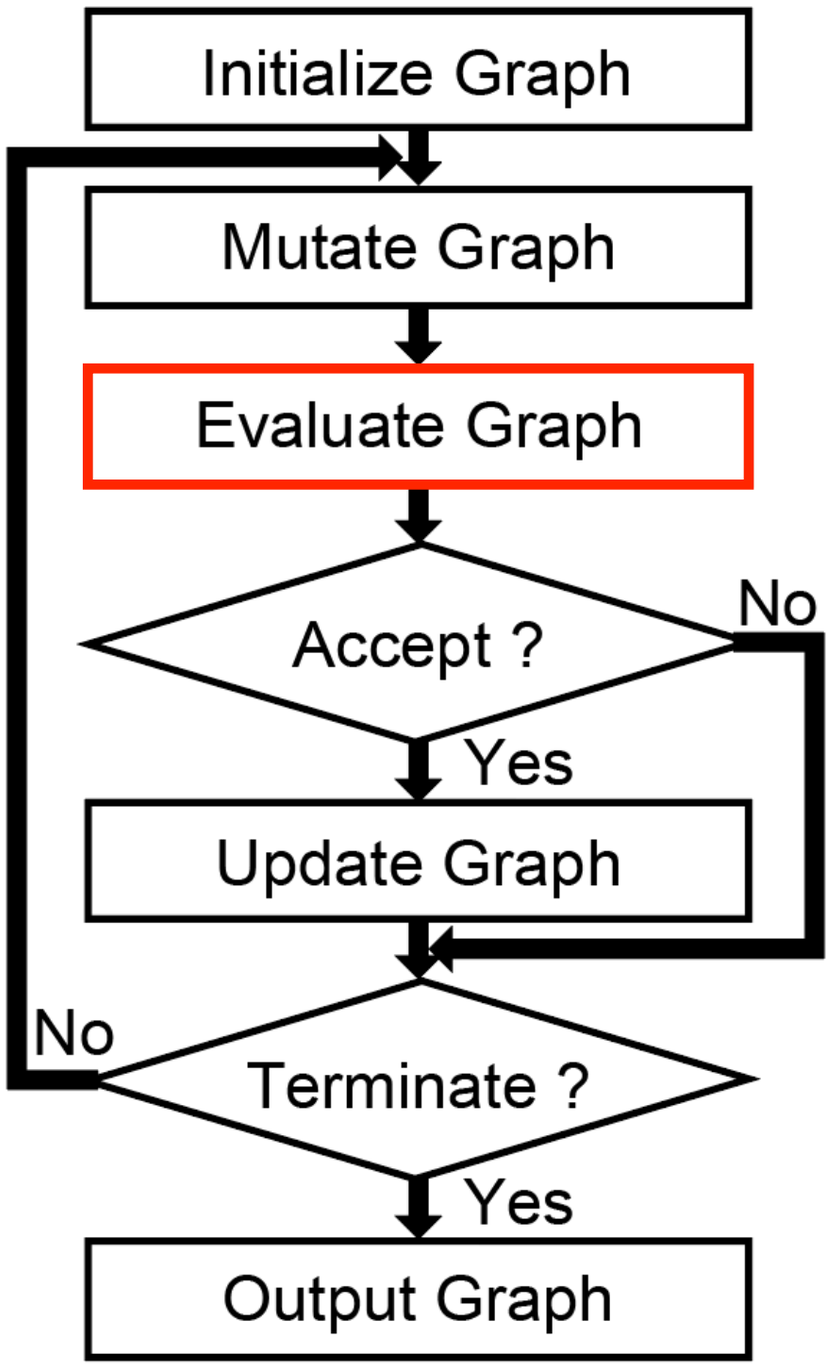


(n, d) = (10, 3)

	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										

diameter = 3
ave. hops ≒ 1.89

Background (3/3)



- To create a random graph with small diameter and average hops, Simulated Annealing (SA) are often used
- SA repeats **APSP calculation many times**
  - **However, the calculation cost of APSP is very high !!**
- The complexity is proportional to the square of the number of vertices (n)

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 Xeon Gold 6126

Objective and a part of results

- Our previous research in HPC Asia 2019 provides SA algorithm where APSP is calculated using Breadth-First Search (**BFS-APSP**)
  - **BFS-APSP** is parallelized by MPI+OpenMP

- This research introduces another APSP algorithm based on adjacency matrix (**ADJ-APSP**), and compares with **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 MPI+OpenMP on 64 CPUs x 12 Cores (6.77 sec.) →
ADJ-APSP by MPI+CUDA on 128 GPUs (**0.28 sec.**)

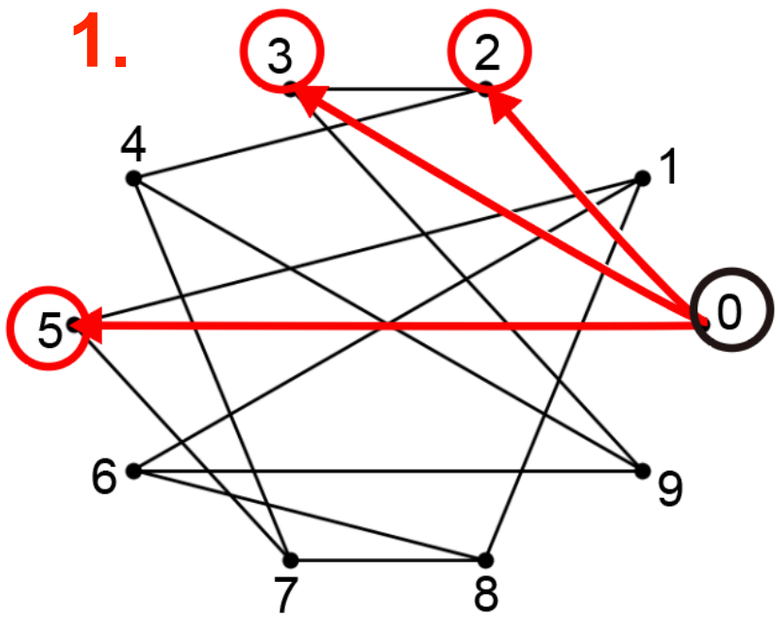
You can download programs from
https://github.com/mnakao/APSP/

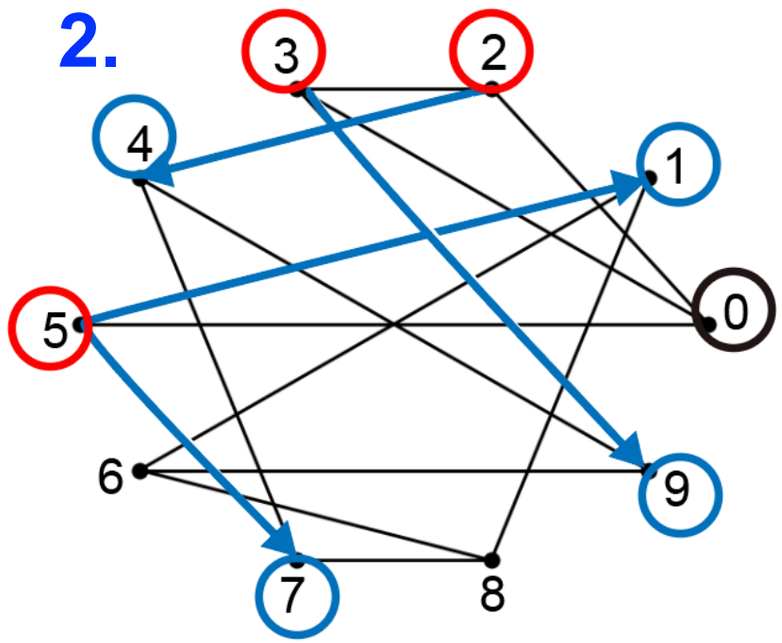
Agenda

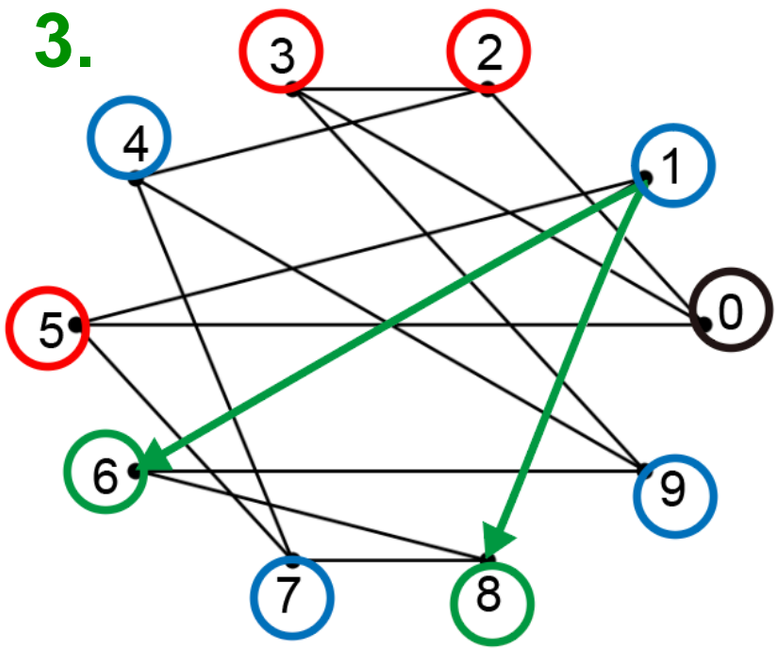
- Background
- BFS-APSP
- ADJ-APSP
- Performance
- Summary

Serial BFS-APSP

- BFS can find the hops from one vertex to others
  - APSP can be obtained by executing BFS from all vertices
  - Top-down approach is used
- The computational complexity of BFS is O(nd).
  - The computational complexity of BFS-APSP is $O(n^2d)$ because BFS is repeated n times







Parallel BFS-APSP

- Multiple BFSs are performed in parallel using MPI from different vertices, and one BFS is executed using OpenMP
- MPI parallelization,
  - Starting points are assigned to each MPI process evenly
  - Thus, the maximum number of MPI processes is n
  - Communication time is small because only scalar values of diameter and average hops in each MPI process are collected at the end of the program
- OpenMP parallelization,
  - Each OpenMP thread searches not-visited vertices
  - The implementation requires exclusive control to update own list of not-visited vertices

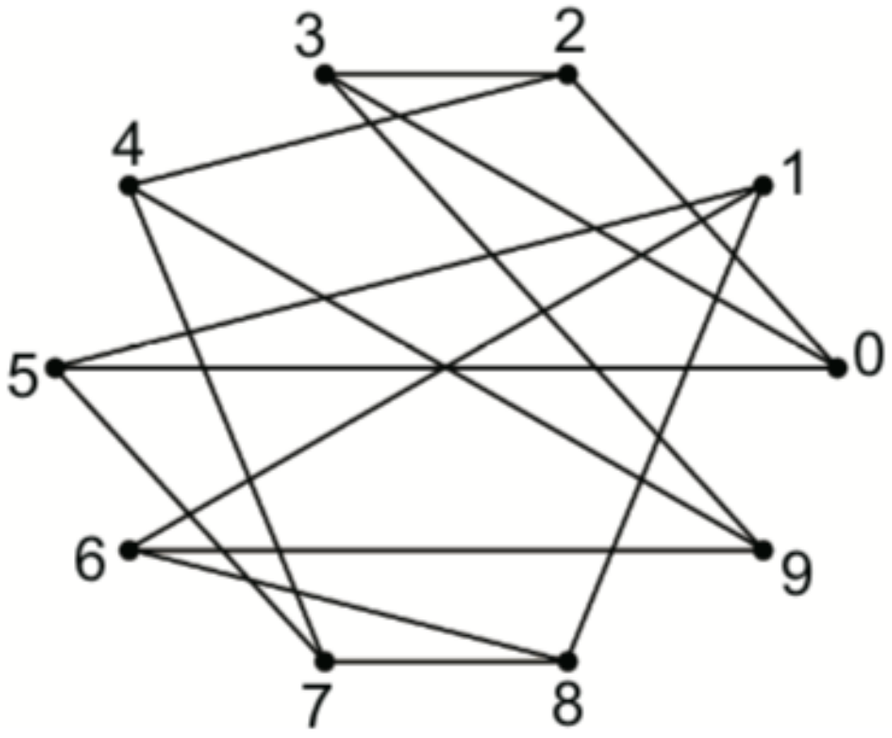
Agenda

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

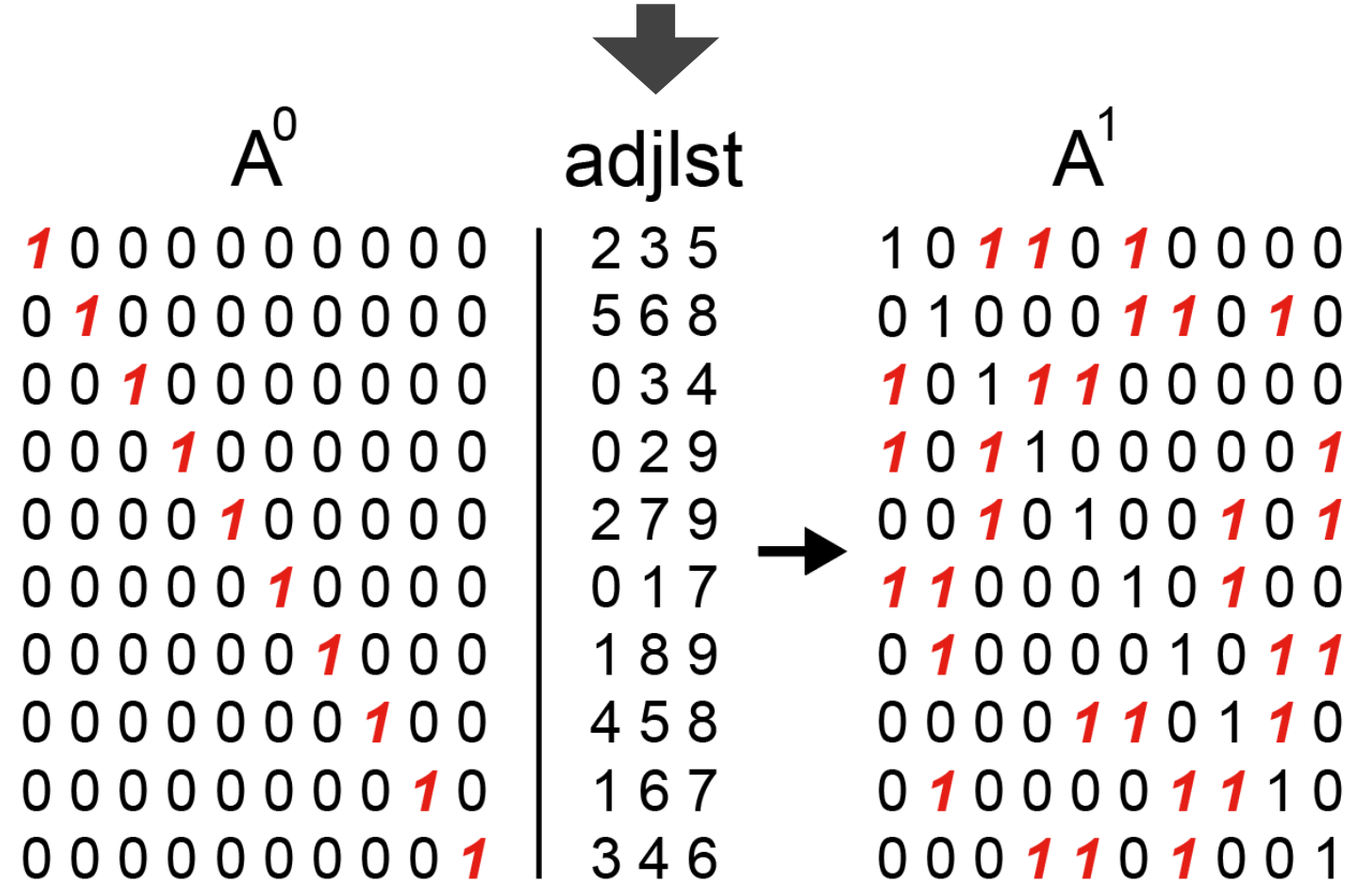
Serial ADJ-APSP(1/3)

- 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

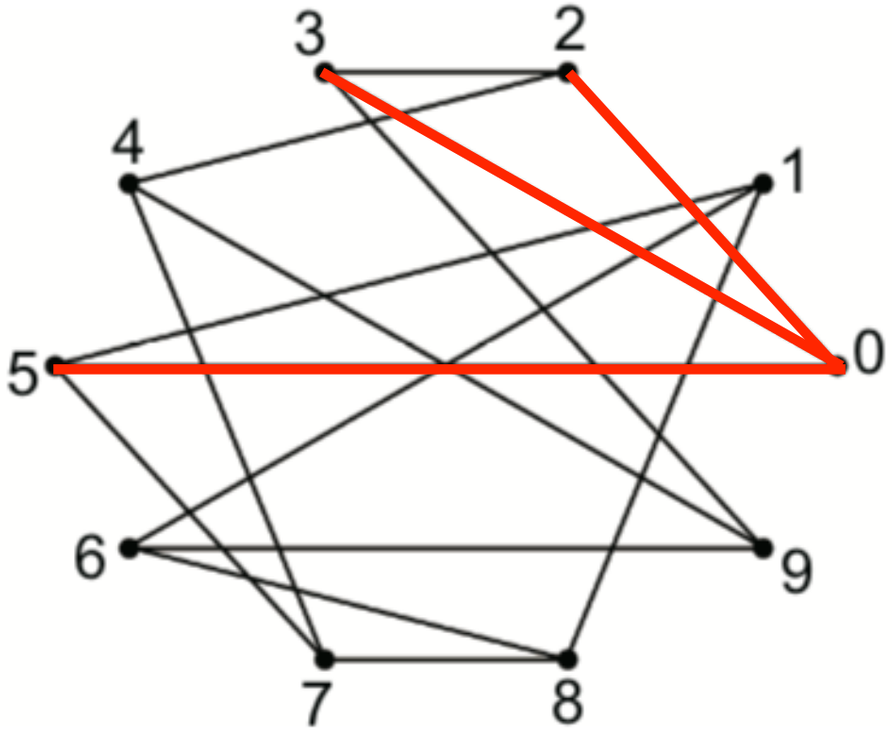


Serial ADJ-APSP(1/3)

- 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^0 | adjlst | A^1

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

Serial ADJ-APSP(2/3)

- When all elements are 1, the value of k is Diameter
- Average hops can be calculated by summing all elements whose value for a_{i, j} is 0 divided by number of elements (170/90 ≒ 1.89)

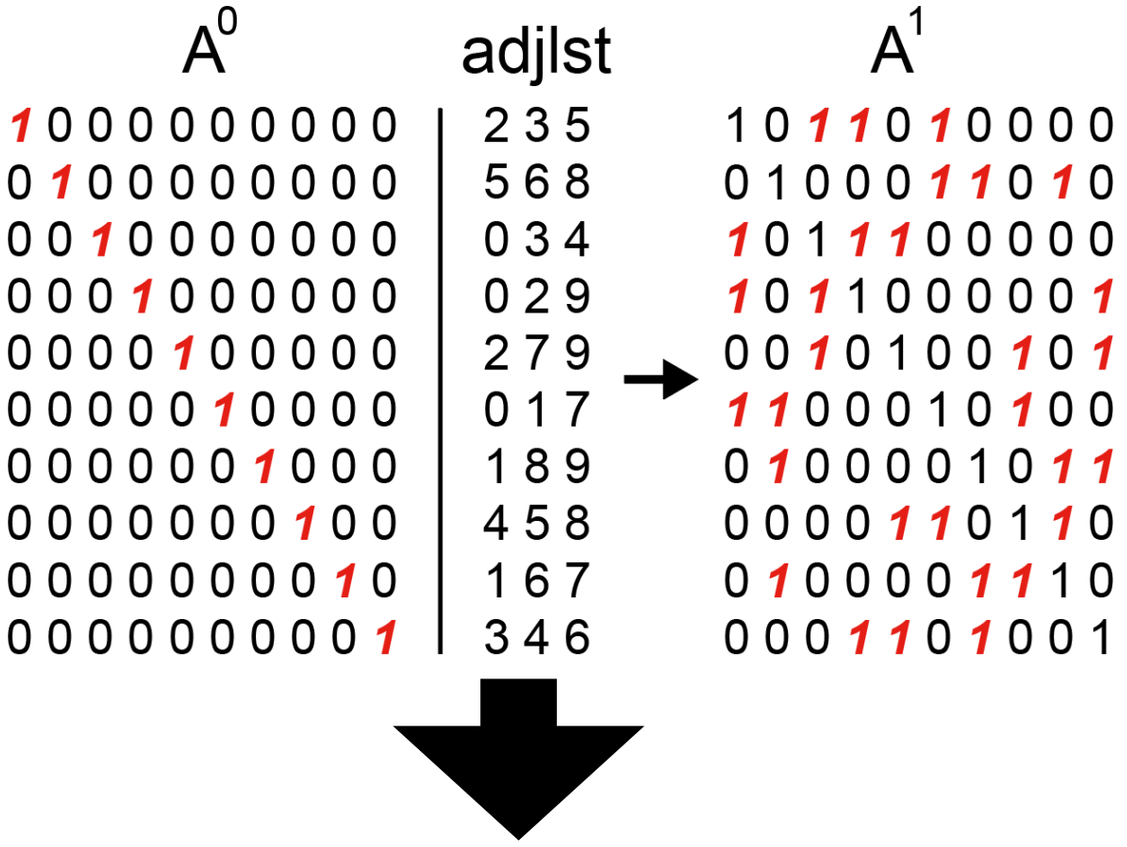
Serial ADJ-APSP(3/3)

```
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
```

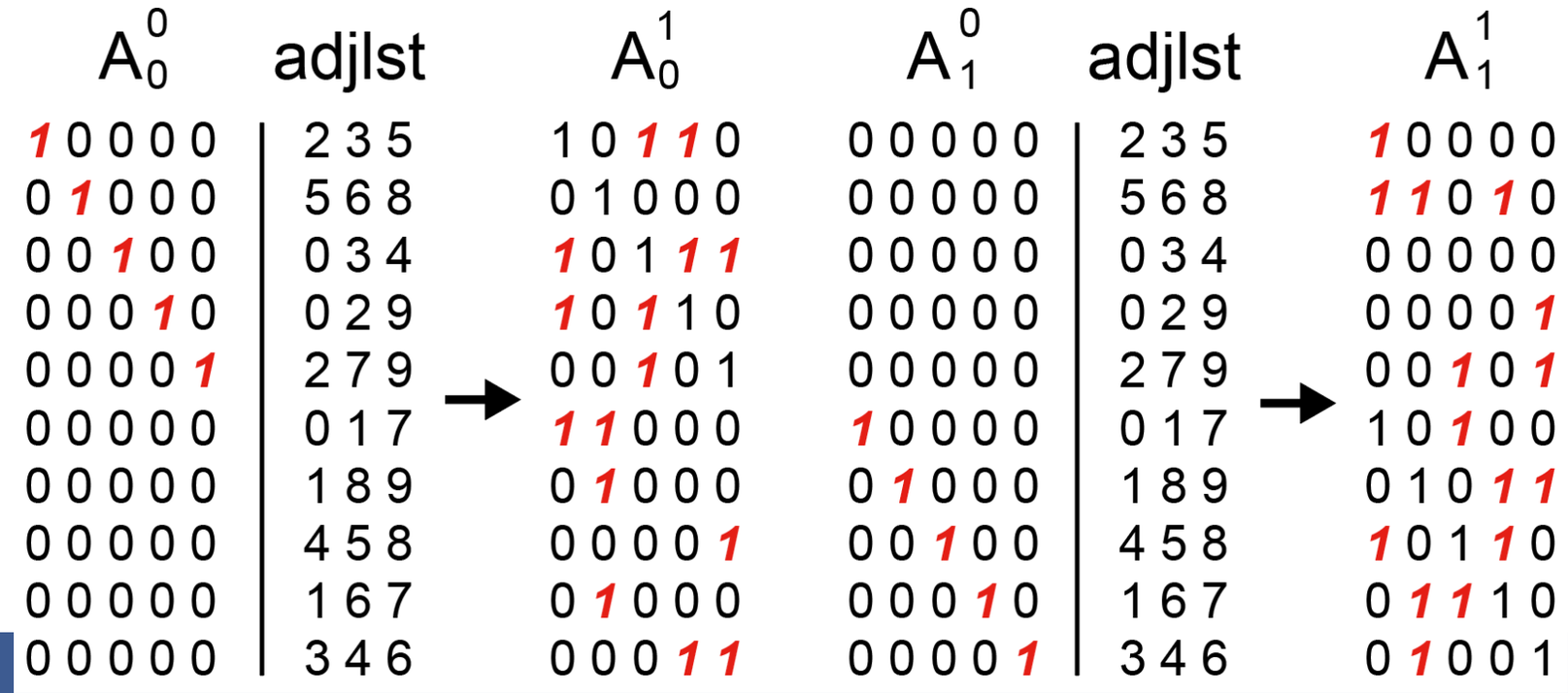
← Bit OR operation
(the most time-consuming part)

←__builtin_popcountll() or _mm_popcnt_u64()

Parallel ADJ-APSP



By dividing "A" vertically,
"A" can be calculated independently by MPI



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
```

Dividing "A" vertically

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() for scalar x 2	

n: vertices
d: degree
D: diameter *
E: bits in element (64)
(we use uint64_t)

* In general, the value of D of graphs with randomness is small due to the small-world effect.

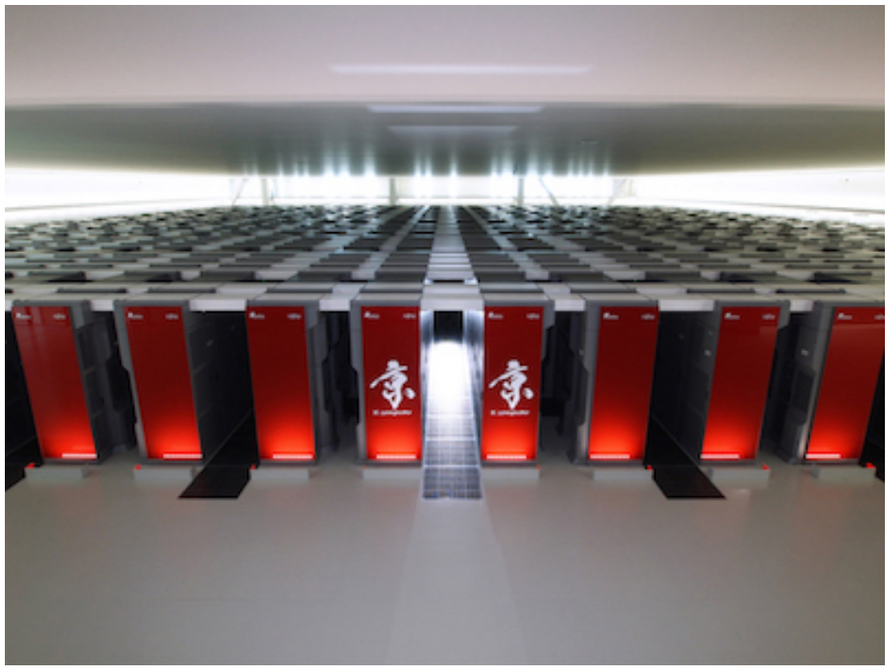
** The number of elements of columns in an adjacency matrix

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 MPI+OpenMP versions

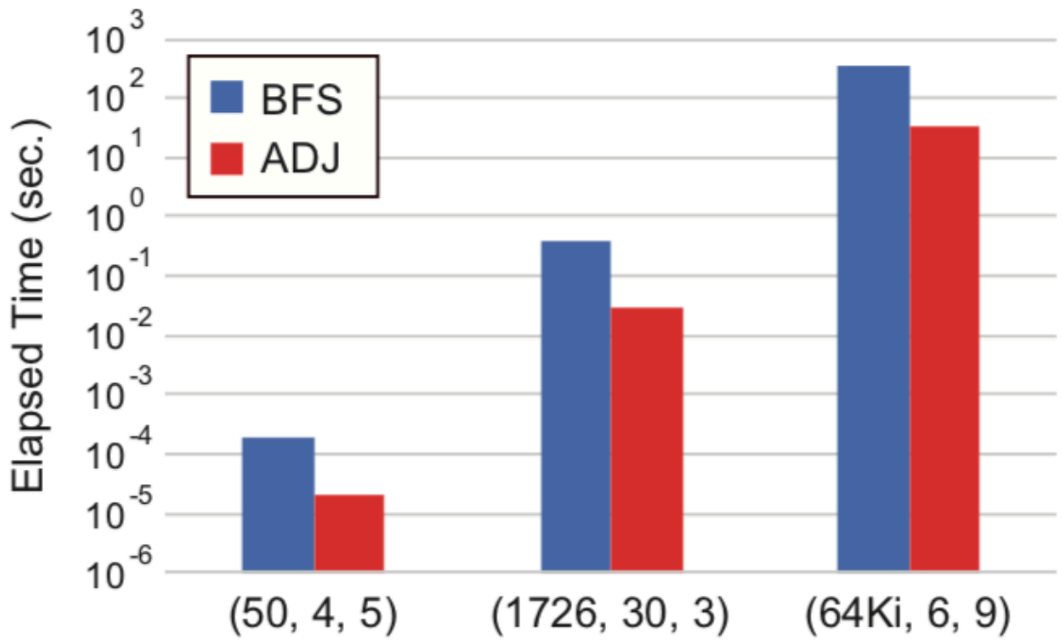
Cygnus in CCS, Univ. of Tsukuba

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

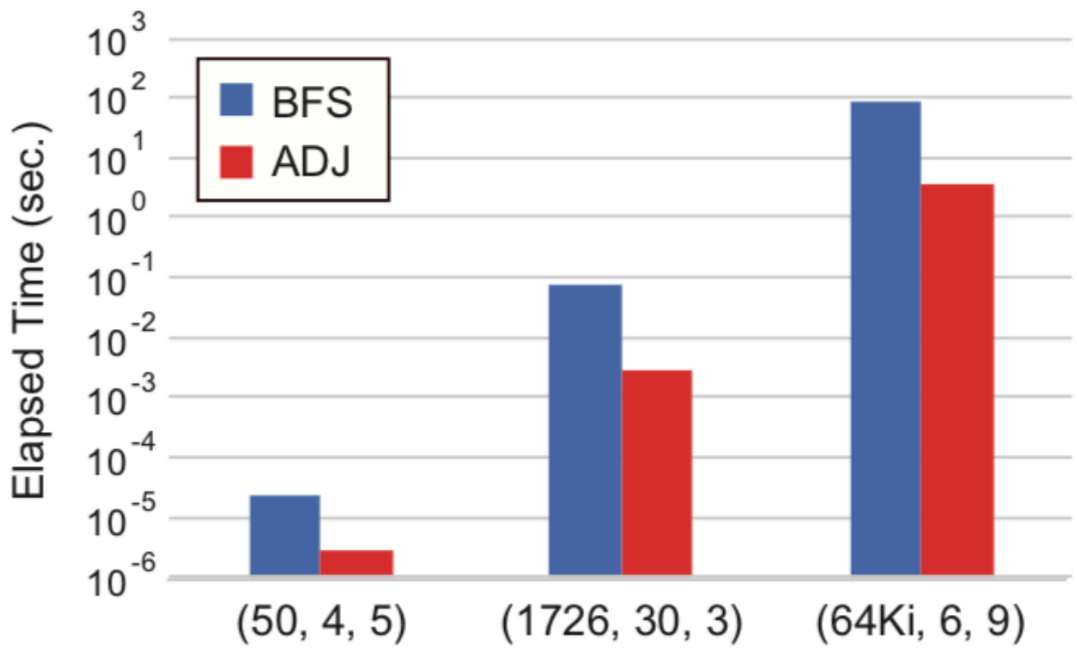
For MPI+OpenMP versions
For MPI+CUDA version

Serial algorithm

The K computer



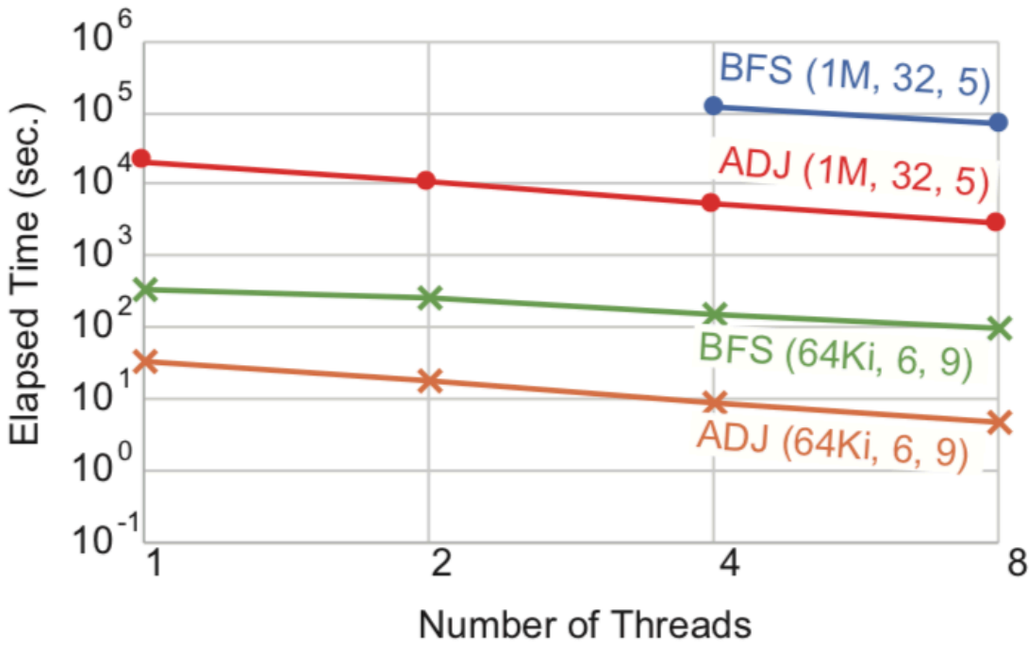
Cygnus system



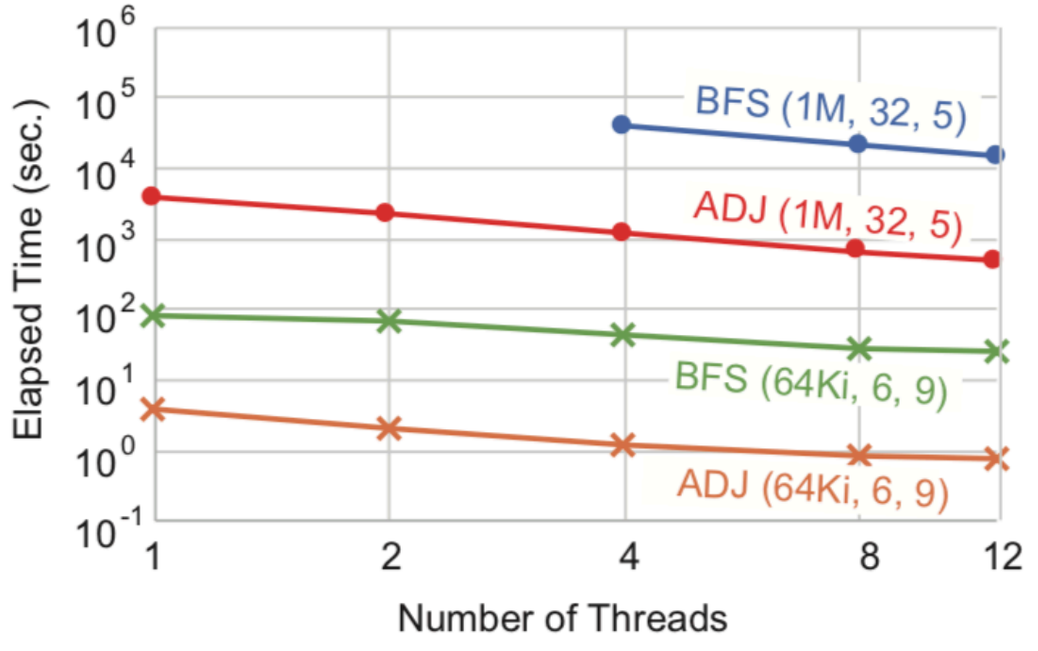
- (n, d, D) = (50, 4, 5), (1726, 30, 3), and (64Ki, 6, 9) *64Ki is 65,536.
- Test programs are available at http://research.nii.ac.jp/graphgolf/
- ADJ-APSP is always faster than BFS-APSP
  - 8.08 to 29.49 times better

Parallel algorithm by OpenMP

The K computer



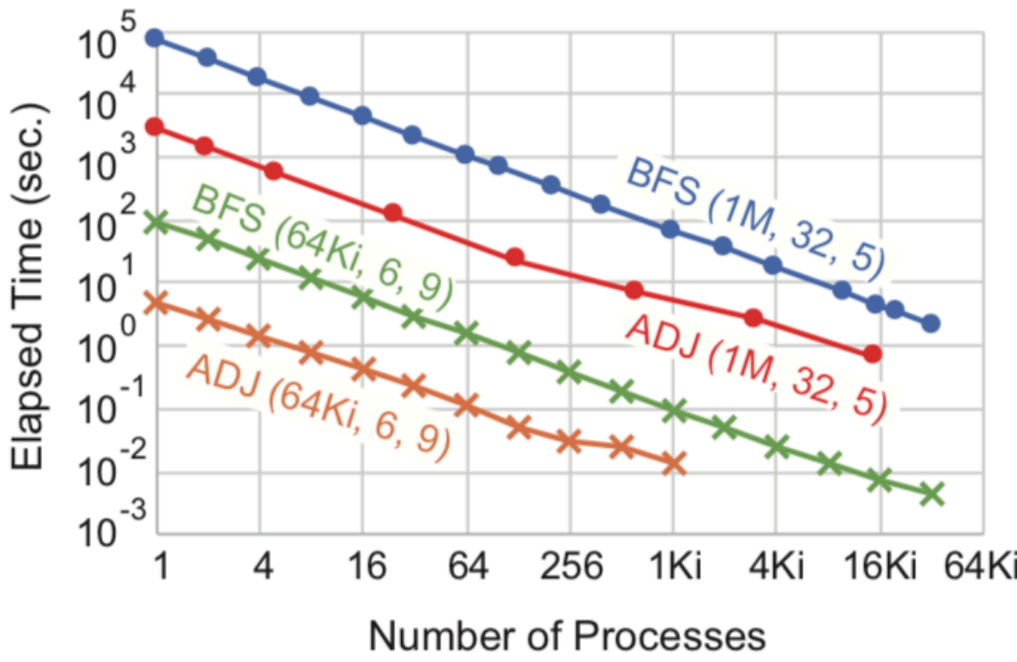
Cygnus system



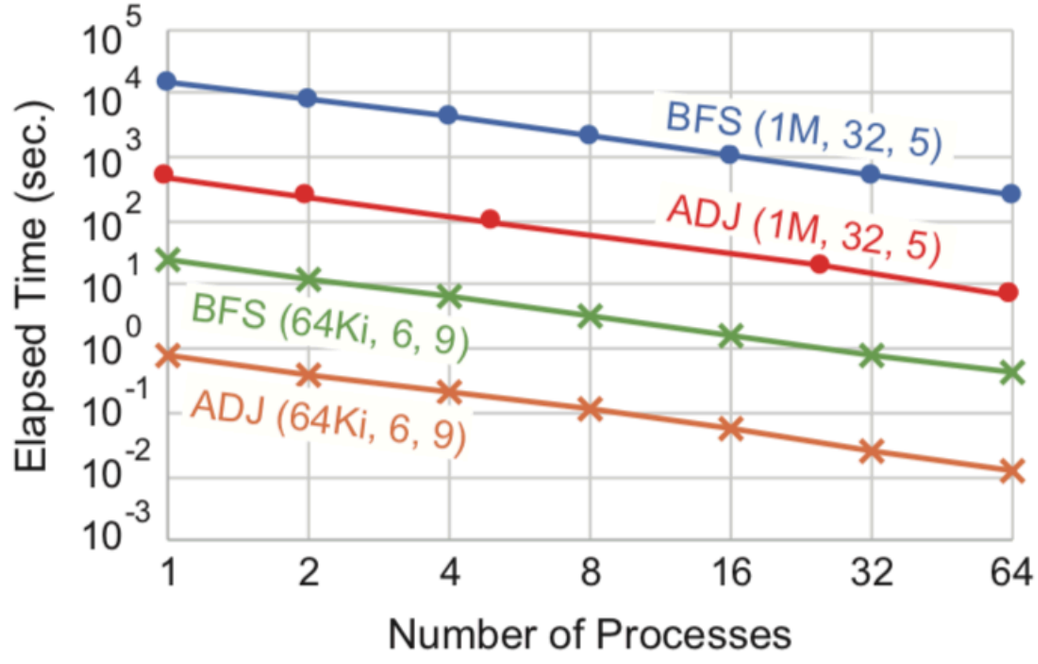
- (n, d, D) = (64Ki, 6, 9) and (1M, 32, 5)
- ADJ-APSP is always faster than BFS-APSP too
  - Improvement ratio of ADJ-APSP is better than that of BFS-APSP
  - In case of (64Ki, 6, 9) using 8 threads on the K computer, improvement ratio of ADJ-APSP is 7.18, and that of BFS-APSP is 3.54
  - In case of (1M, 32, 5) using 12 threads on Cygnus system, ADJ-APSP(1threads) : 3,880sec. → ADJ-APSP(12 threads) : 475sec.

Parallel algorithm by MPI+OpenMP

The K computer



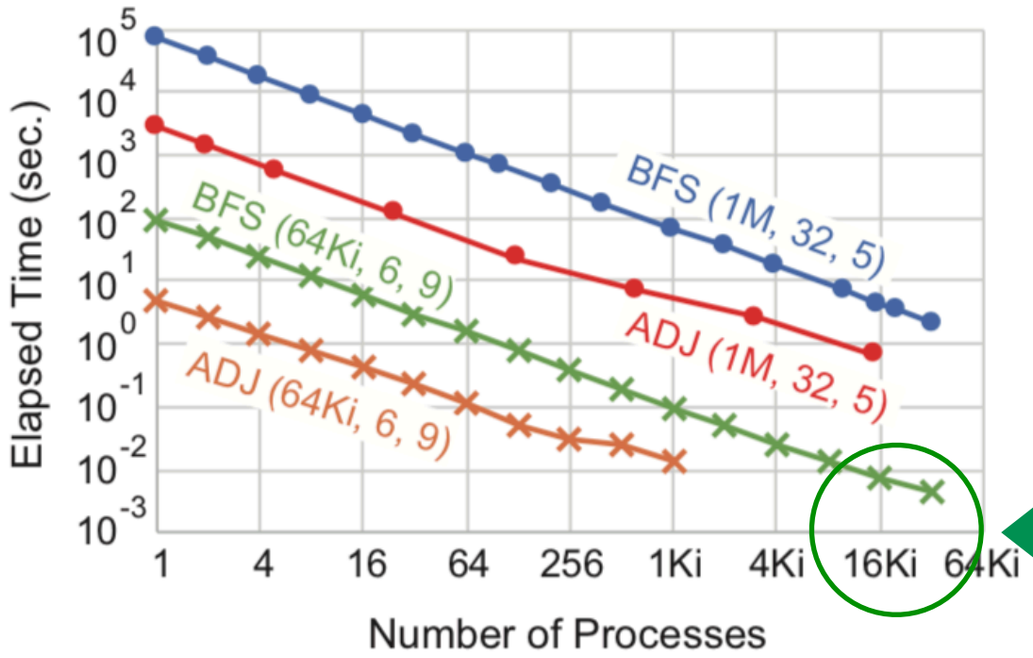
Cygnus system



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

Parallel algorithm by MPI+OpenMP

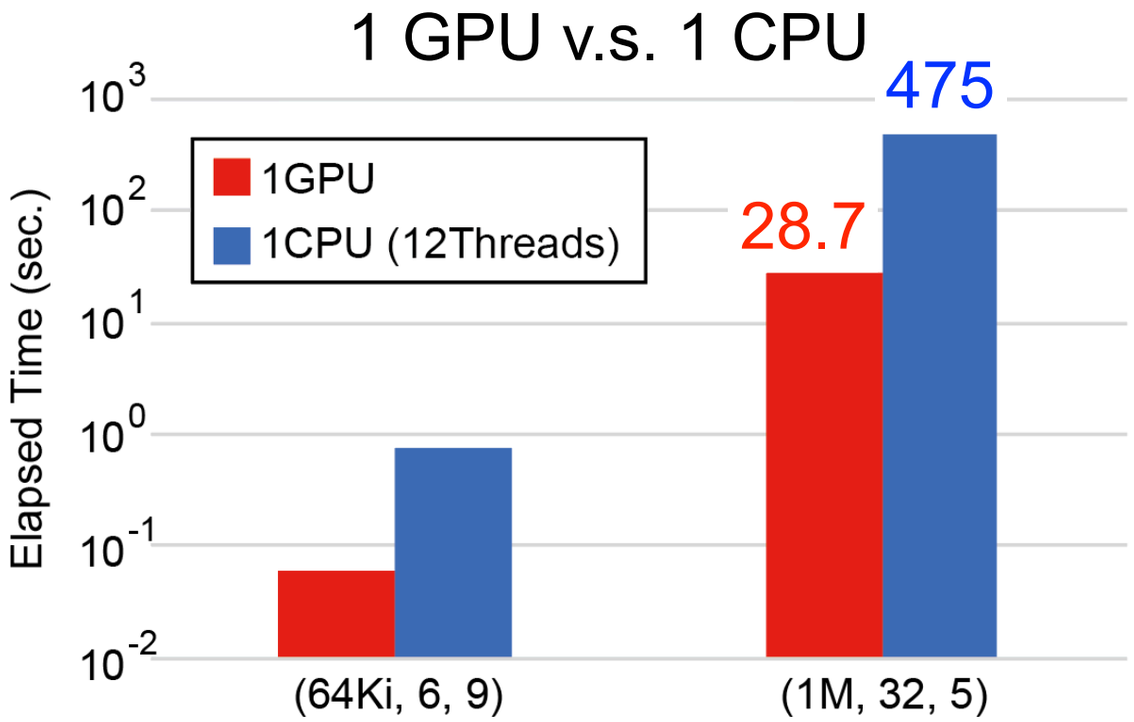
The K computer

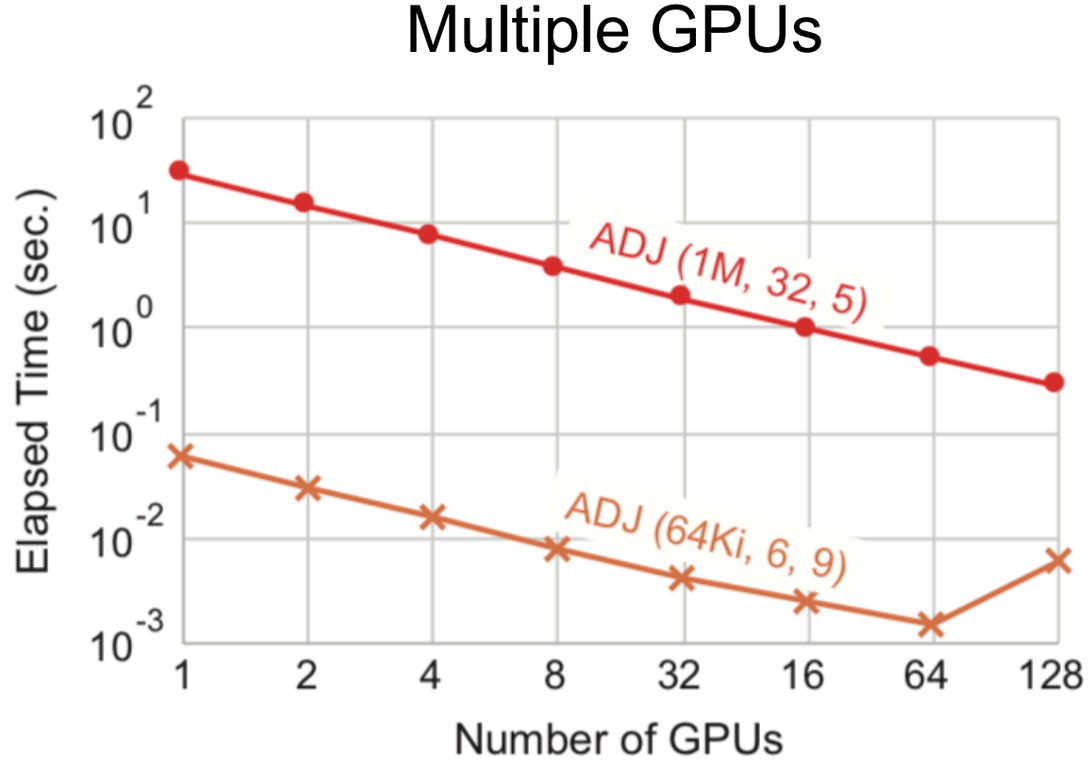


BFS-APSP is faster than ADJ-APSP in a large number of processes

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

Parallel algorithm by MPI+CUDA

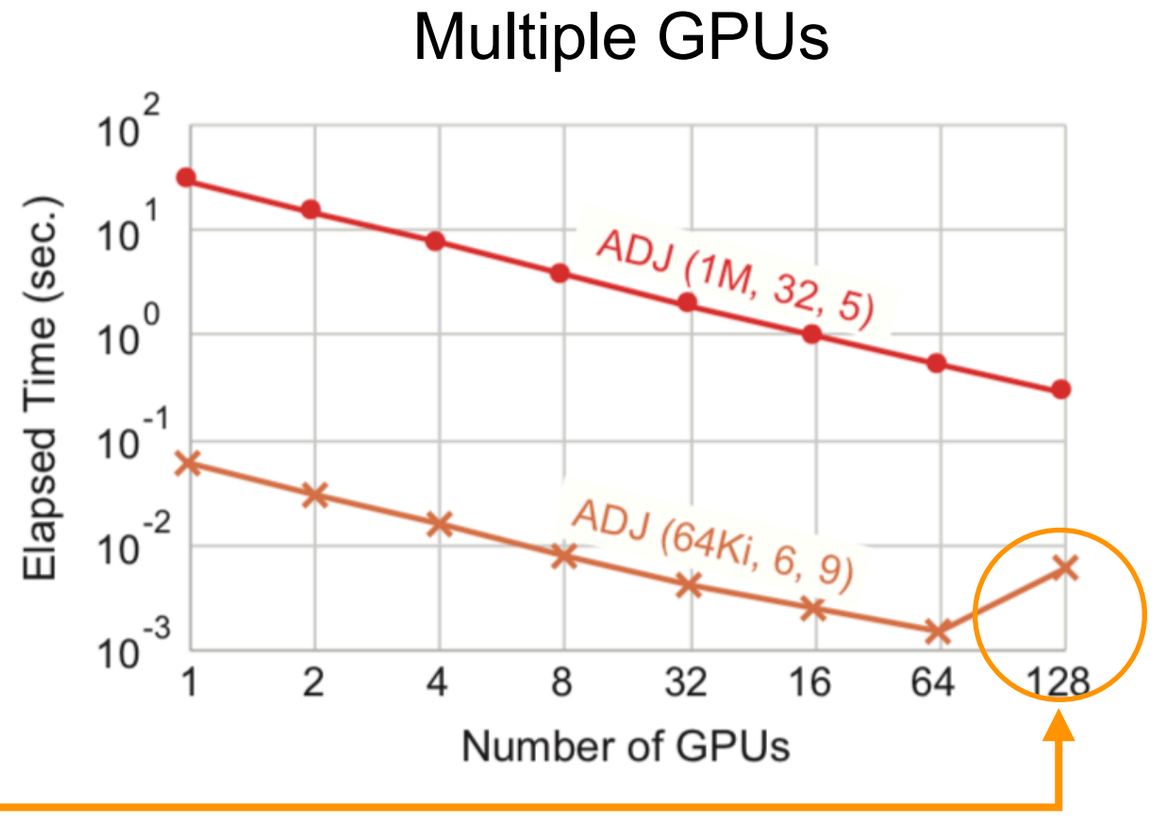




- 1GPU v.s. 1CPU
  - (64Ki, 6, 9): 0.751 sec. (1CPU) → 0.061 sec. (1GPU) x 12.6
  - (1M, 32, 5): 475 sec. (1CPU) → 28.7 sec. (1GPU) x 16.5
- Multiple GPUs
  - (64Ki, 6, 9): 0.0608 sec. (1GPU) → 0.0015 sec. (64GPUs) x 38.4
  - (1M, 32, 5): 28.7 sec. (1GPU) → 0.28 sec. (128GPUs) x 101.1

Parallel algorithm by MPI+CUDA

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



- 1GPU v.s. 1CPU
  - (64Ki, 6, 9): 0.751 sec. (1CPU) → 0.061 sec. (1GPU) x 12.6
  - (1M, 32, 5): 475 sec. (1CPU) → 28.7 sec. (1GPU) x 16.5
- Multiple GPUs
  - (64Ki, 6, 9): 0.0608 sec. (1GPU) → 0.0015 sec. (64GPUs) x 38.4
  - (1M, 32, 5): 28.7 sec. (1GPU) → 0.28 sec. (128GPUs) x 101.1

Agenda

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

Summary

- We parallelize BFS-APSP and ADJ-APSP using MPI+OpenMP and MPI+CUDA
- 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 a large computational resource may be better than that of ADJ-APSP
- We achieved further speedup by parallelizing ADJ-APSP using GPUs
  - 28.7sec. (ADJ, 1GPU) → 0.28 sec.(ADJ, 128GPUs)