Heterogeneous Parallel Programming COMP4901D

All-Pair Shortest Path Search on a GPU Cluster

Overview

- All-Pair Shortest Path Problem
- Existing Sequential Algorithms
- Existing Parallel Algorithms
- Extended GPU-based Algorithm

All-Pair Shortest Path Search

- Given a weighted graph G(V,E,W), all-pairs shortest path (APSP) finds paths with minimum edge weights between all pairs of vertex u and v.

- Applications: Social network, routing network, transportation network

Floyd-Warshall

Dynamic Programming

- $P(i, j, k)$: Shortest paths from *i* to *j* with intermediate vertices label in the set {1…k}.
- Base case: $P(i, j, 0) = w(i, j)$
- Recursive case: $P(i, j, k+1) = \min\{P(i, j, k), P(i, k+$

Floyd-Warshall(cont)

Algorithm

- for k=0 to |V|
- for i=0 to |V|
- for j=0 to |V|
- $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$

Complexity

- $O(V^3)$

Dijkstra

Greedy Approach

- Set *S* stores the vertices whose shortest paths have been determined.
- Iteratively choose a vertex with the current smallest path cost value to insert into *S.*

Data Structure

- Minimum-priority queue *Q*

Dijkstra(cont)

Algorithm

- While Q is not empty
- do $u \leftarrow Extract\_\min(Q)$
- $S \leftarrow S \cup \{u\}$
- for each vertex $v \in adj(u)$
- $D(v) = \min\{D(v), D(u) + w(u, v)\}$

Complexity

- $O(E + VlogV)$ for *SSSP*
- $O(VE + V^2logV)$ for *APSP*

Bellman-Ford

Compared with Dijkstra

- Drawback: Slow
- Advantage: Allow edges with negative weights.

Algorithm

- for i=0 to |V|-1
- for each edge $(u, v) \in E$
- $D(v) = \min\{D(v), D(u) + w(u, v)\}$

Complexity

- $O(VE)$ for *SSSP*
- $O(V^2E)$ for *APSP*

Blocked Floyd-Warshall

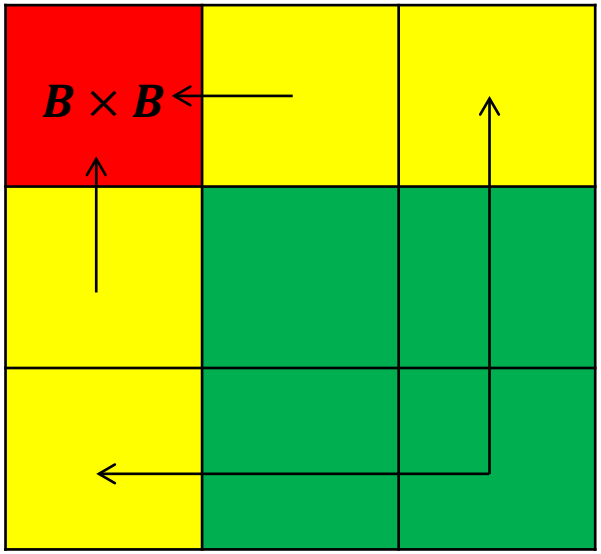
Block Idea

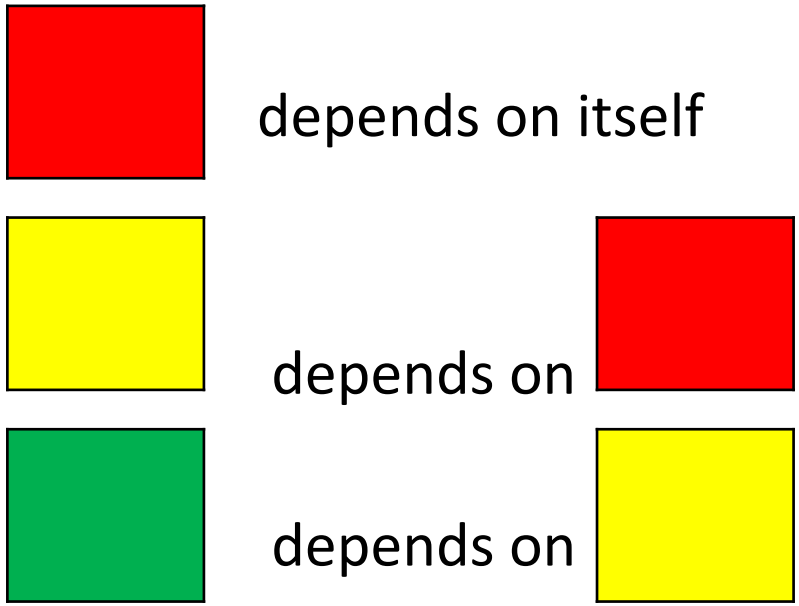
- Utilize dependency of entries in adjacency matrix while running the Floyd-Warshall algorithm.
- Divide the adjacency matrix into blocks with equal size of $B \times B$ and each block runs B iterations of Floyd-Warshall algorithm at each round.
- In each round, process is divided into three phases. Blocks run within a same phase are independent with each other.
- Improve performance by reducing cache miss rate.

Blocked Floyd-Warshall (cont)

Dependency(first round)

- $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- k: 0 to B-1

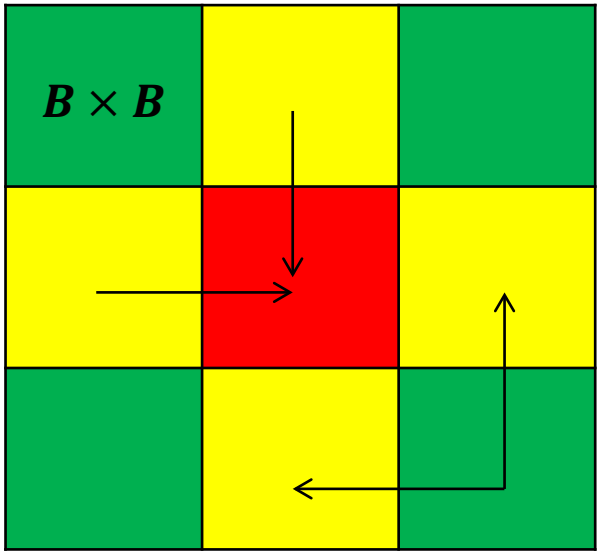


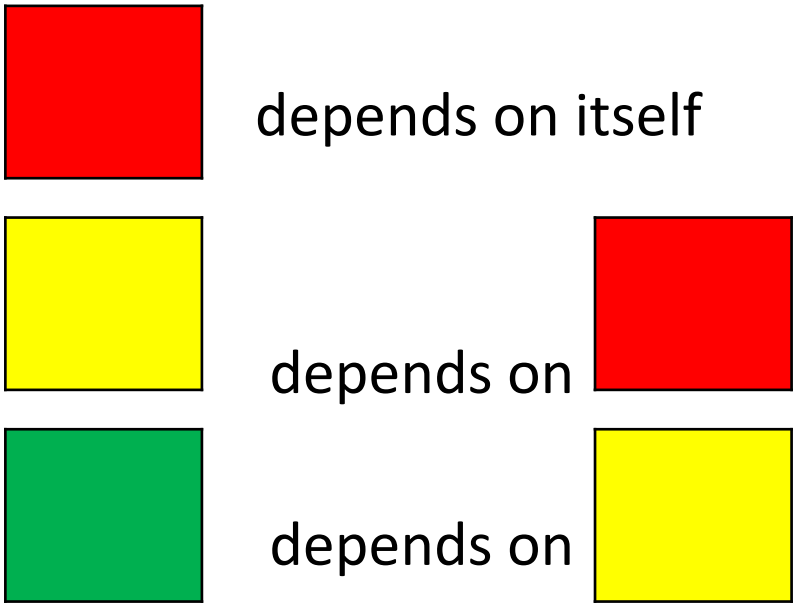


Blocked Floyd-Warshall (cont)

Dependency(second round)

- $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- k: B to $2B$-1

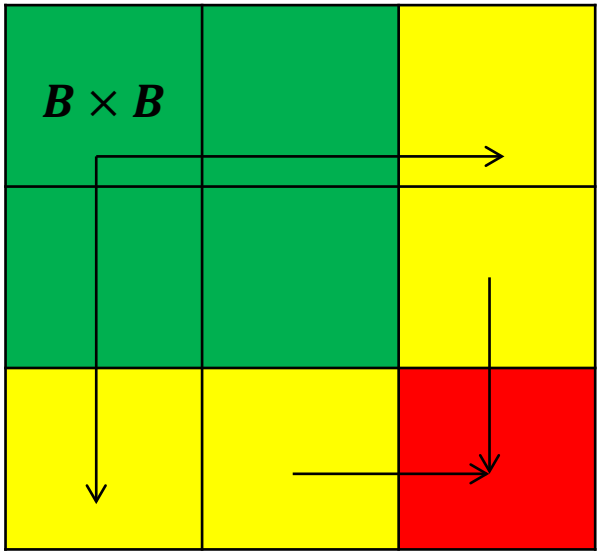


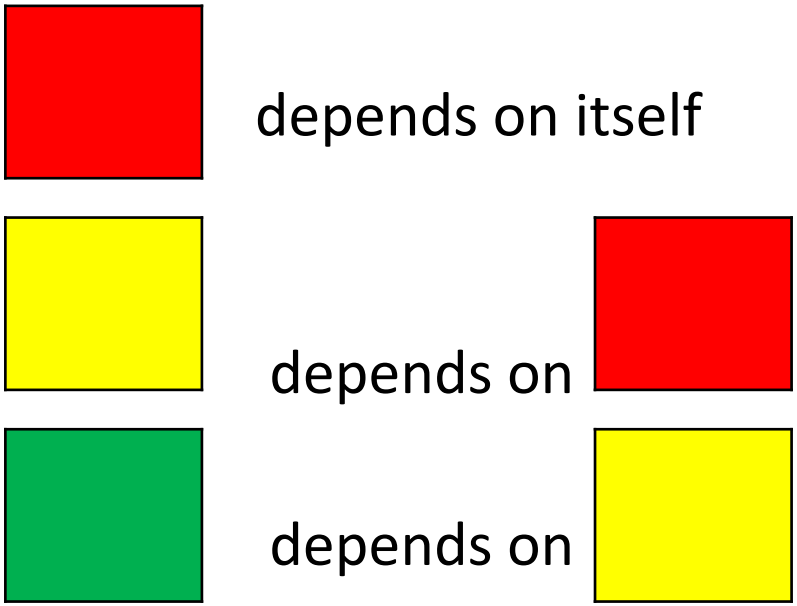


Blocked Floyd-Warshall (cont)

Dependency(last round)

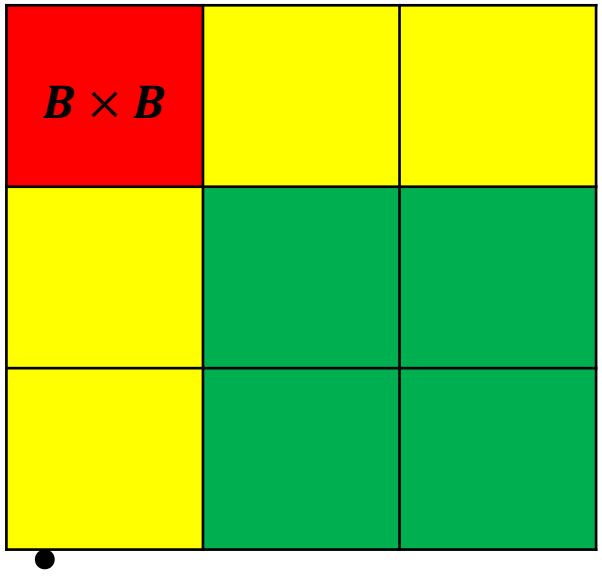
- $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- k: $2B$ to $3B$-1





Blocked Floyd-Warshall (cont)

Three phases(first round)



phase 1:

phase 2:

phase 3:

Blocked Floyd-Warshall (cont)

Algorithm

- for *round*=0 to $\frac{|V|}{B}$-1
- for *k*=*round***B* to (*round*+1)**B*-1
- for all i,j in active block of phase 1
- $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- do the same thing for active blocks of phase 2
- do the same thing for active blocks of phase 3

Complexity

- $O(V^3)$

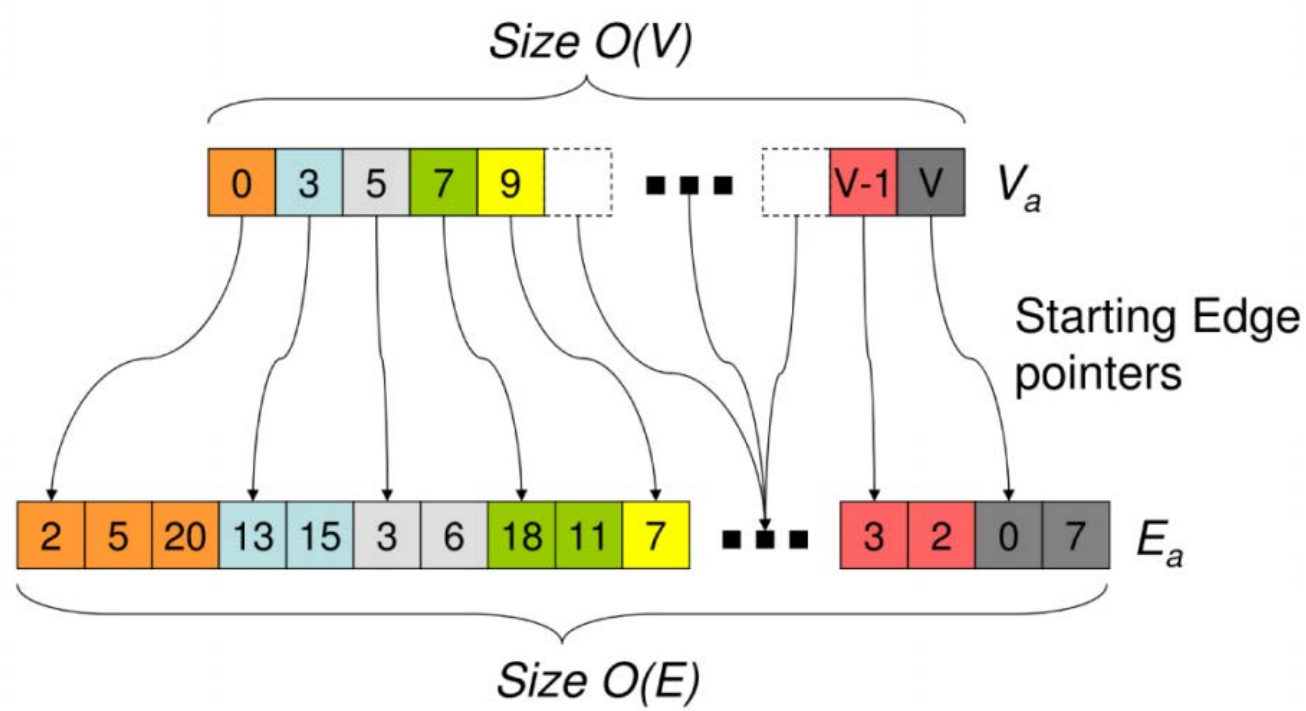
P. Harish's Algorithm

Idea

- Based on Dijkstra's algorithm.
- Map one vertex to each thread.
- Use an array *Mask* to identify update information of vertex.
- In each iteration, corresponding edges of vertices whose *Mask* value are true are relaxed in each corresponding thread.

P. Harish's Algorithm (cont)

Graph Representation



P. Harish's Algorithm (cont)

Kernel 1

- tid=getThreadID
- if *Mask*[tid]==True
- *Mask*[tid]=False
- for all neighbours nid of tid
- if *U*[nid]>*C*[tid]+*W*[nid]
- *U*[nid]=*C*[tid]+*W*[nid]

Kernel 2

- If *C*[tid]>*U*[tid]
- *C*[tid]=*U*[tid]
- *Mask*[tid]=True
- *U*[tid]=*C*[tid]

Floyd-Warshall Algorithm using CUDA

FW_CUDA(*G*)

- Create Adjacency matrix *A* from *G.*
- for k=1 to |V|
- FW_CUDA_KERNEL(*A*,k)

FW_CUDA_KERNEL(*A*,k)

- i=getThreadIdx
- j=getThreadIdy
- if *A*(i,j)>*A*(i,k)+*A*(k,j)
- *A*(i,j)=*A*(i,k)+*A*(k,j)

Quoc-Nam Tran's First Algorithm

Idea

- Use two identical adjacency matrix to avoid global memory communication.
- Each kernel runs one iteration of Floyd-Warshall algorithm.
- Utilize high-bandwidth shared memory.

Shared Memory

- *A*(i,j), *A*(i,k), *A*(k,j).
- Each block in kernel copies *A*(i,k) and *A*(k,j) into shared memory.
- Threads whose row ID=0 copies row *A*(k,j) and column ID=0 copies column *A*(i,k).

Quoc-Nam Tran's First Algorithm(cont)

Tran_CUDA_FW(*A*,*D*,|V|)

- for k=1 to |V|
- if k%2==0
- Tran_FWKernel(*A*,*D*,k)
- else Tran_FWKernel(*D*,*A*,k)

Tran_FWKernel(*A*,*D*,k)

- tid=getLocation(bx,by,tx,ty)
- row[16], col[16] // shared
- if ty==0
- row[tx]=*A(k,j)*
- If tx==0
- col[ty]=*A(i,k)*
- *D*[tid]=min{*A*[tid], row[tx]+col[ty]}

Quoc-Nam Tran's Second Algorithm

Idea

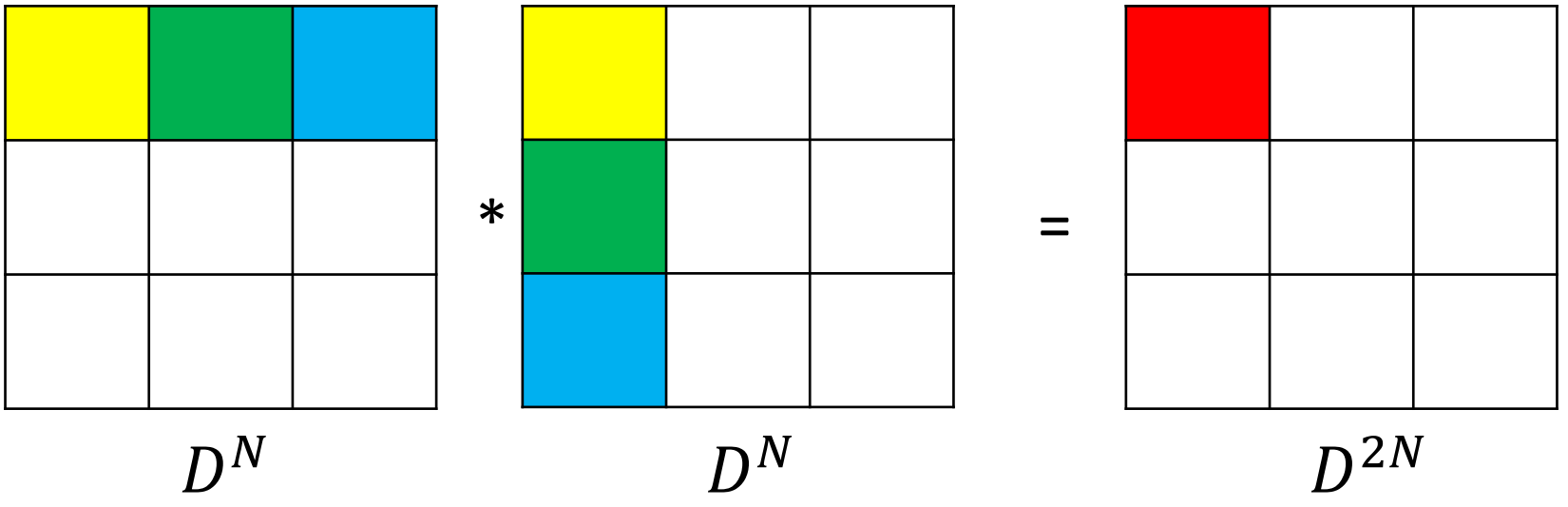
- D^N stores shortest path cost values with length at most *N*.
- $D^2 = \min\{D^1[i,k] + D^1[k,j]\}, where\ D^1 = A\ and\ k \in V$
- $D^{|V|}$ stores final values.
- Utilize D^1 to construct D^2, D^4, … , $D^{|V|}$.

Kernel

- $D^{2N}(i,j)$ is calculated by i^{th} row and j^{th} column of D^N.
- Each block copies one row tile of D^N and one column tile of D^N into shared memory in each iteration.
- Process similar to matrix multiplication.

Quoc-Nam Tran's Second Algorithm(cont)

- $D^{2N} = \min\{D^N[i,k] + D^N[k,j]\}$



Quoc-Nam Tran's Second Algorithm(cont)

Tran_CUDA_APSP(*A*,*D*,|V|)

- for k=1 to log(|V|-1)
- if k%2==0
- Tran_APSPKernel(*A*,*D*,k)
- else Tran_APSPKernel(*D*,*A*,k)

Tran_APSPKernel(*A*,*D*,k)

- tid=getLocation(bx,by,tx,ty)
- minvalue=∞
- *D1*[16][16], *D2*[16][16] //shared
- for each tile *D1* and *D2* in A
- minvalue=min{minvalue, *D1*[ty,k]+*D2*[k,tx]}
- *D*[tid]=minvalue

Katz and Kider's algorithm

Idea

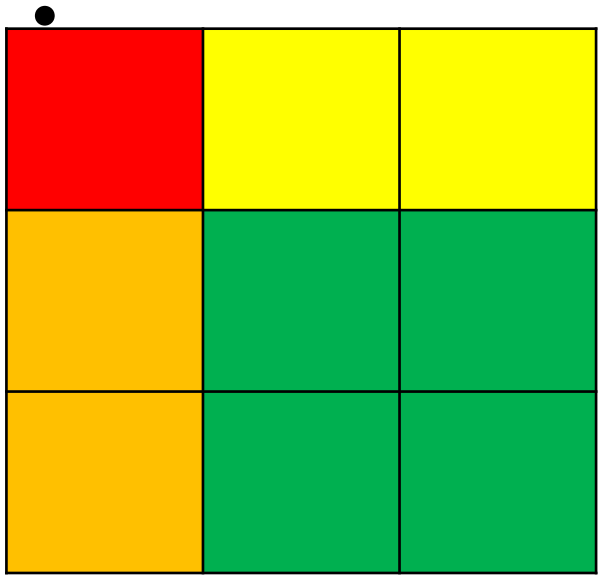
- Based on blocked Floyd-Warshall algorithm.
- Divide adjacency matrix into blocks which fit in the thread blocks of GPU.
- Three kernels with one for each phase.

Kernels

- Suppose adjacency matrix is divided into N×N.
- 1 active block in kernel 1.
- 2(N-1) active blocks in kernel 2.
- (N-1)(N-1) active blocks in kernel 3.
- Blocks in each kernel loads corresponding blocks of adjacency matrix into their shared memory.

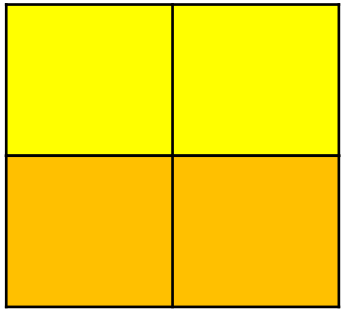
Katz and Kider's algorithm(cont)

Active Blocks in Three Kernels(first round)

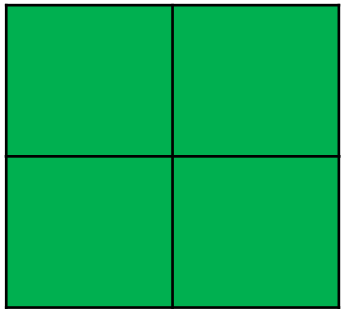


kernel 1:

kernel 2:



kernel 3:



Katz and Kider's algorithm(cont)

KK_GPU1(D,|V|)

- for round=0 to |V|/B
- KK_GPU1_Kernel1(D,round,|V|)
- KK_GPU1_Kernel2(D,round,|V|)
- KK_GPU1_Kernel3(D,round,|V|)

KK_GPU1_Kernel1(D,round,|V|)

- tid=getLocation(tx,ty,round)
- gD[B][B] //Load block into shared memory
- for k=0 to B
- gD[ty][tx]=min{gD[ty][tx], gD[ty][k]+gD[k][tx]}
- Synchronize
- D[tid]=gD[ty][tx]

Katz and Kider's algorithm(cont)

KK_GPU1_Kernel2(D,round,|V|)

- tid=getLocation(bx,by,tx,ty,round)
- gD[B][B], Pri[B][B] //Load two blocks in shared memory
- if by==0
- for k=0 to B
- gD[ty][tx]=min{gD[ty][tx], Pri[ty][k]+gD[k][tx]}
- Synchronize
- else
- for k=0 to B
- gD[ty][tx]=min{gD[ty][tx], gD[ty][k]+Pri[k][tx]}
- Synchronize
- D[index]=gD[ty][tx]

Katz and Kider's algorithm(cont)

KK_GPU1_Kernel3(D,round,|V|)

- tid=getLocation(bx,by,tx,ty,round)
- gD[B][B], Row[B][B], Col[B][B] //Load three blocks in shared memory
- for k=0 to B
- gD[ty][tx]=min{gD[ty][tx], Row[ty][k]+Col[k][tx]}
- Synchronize
- D[index]=gD[ty][tx]

GPU algorithms for APSP

Comparisons

Algorithm	Based On	Utilize SM	Complexity
P. Harish	Dijkstra	No	$O(VE + V^2 logV)$
FW	Floyd-Warshall	No	$O(V^3)$
Tran's First	Floyd-Warshall	Yes	$O(V^3)$
Tran's Second	Floyd-Warshall	Yes	$O(V^3 logV)$
KK	Blocked Floyd-Warshall	Yes	$O(V^3)$

Space For Improvement

Time

- Sequential algorithms are too slow
- One GPU is insufficient for large graphs.

Space

- One GPU has only a few gigabytes of device memory
- Graphs of more than 40K with Katz and Kider's algorithm exceeds the GPU memory capacity.

Algorithm to Extend

Katz and Kider's algorithm

- Best performance among GPU algorithms.
- P. Harish's algorithm needs global communication during and after computation of each iteration.
- Floyd-Warshall algorithm on GPU and Tran's two algorithms also need global communication in each iteration and they need total |V| iterations.
- Katz and Kider's algorithm runs *B* iterations at each round. Within each round, blocks in a phase run independently. Communication happens only between two phases. The algorithm only runs in |V|/*B* iterations.

Multiple GPUs in a single node

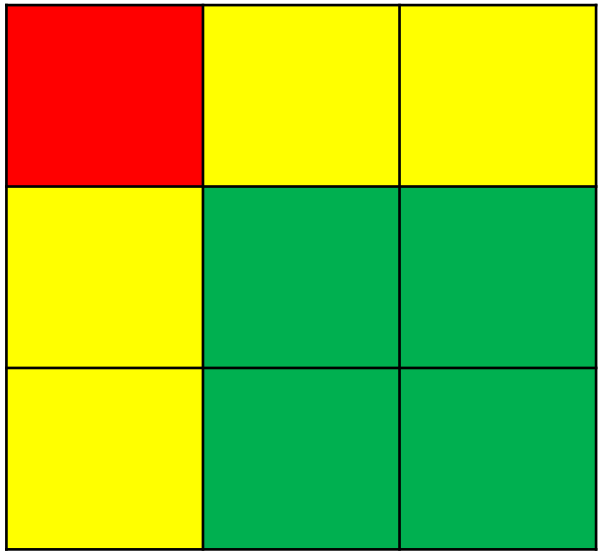
Observation

- Multiple GPUs can run concurrently.
- In KK's algorithm, the calculations of active blocks in each phase are independent.

Idea

- Partition adjacency matrix into blocks.
- Several rounds and three phases in a round.
- Each GPU calculates one active block in certain phase.

Multiple GPUs in a single node(cont)



Adjacency Matrix

Phase 1:

GPU0	GPU1
GPU 2	GPU3

Phase 2:

GPU0	GPU1
GPU 2	GPU3

Phase 3:

GPU0	GPU1
GPU 2	GPU3

Multiple GPUs in a single node(cont)

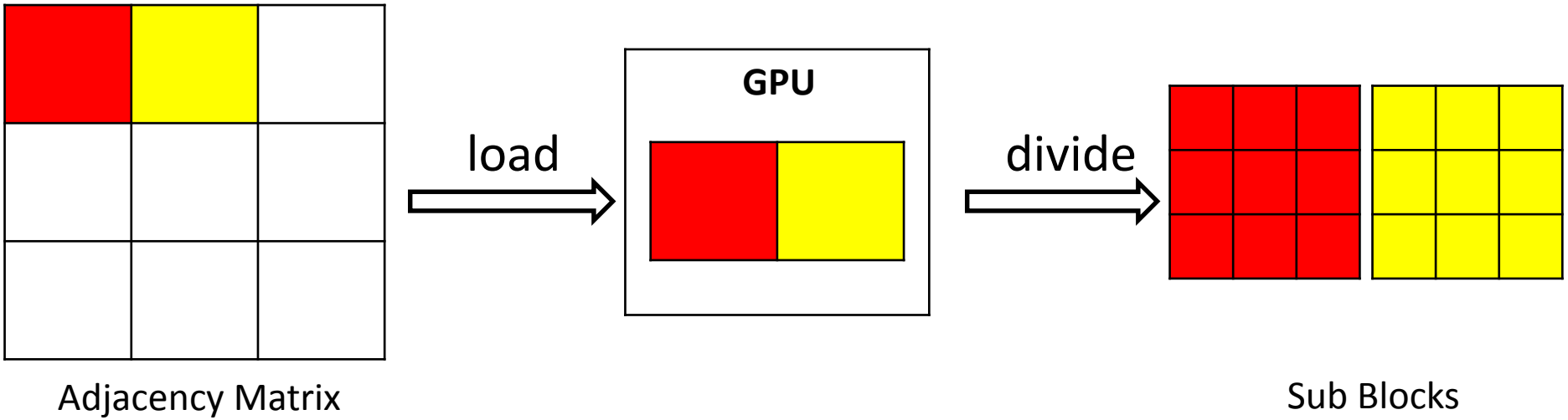
Phase 1

- Only one block and self dependent.
- Run KK's algorithm on a particular GPU.

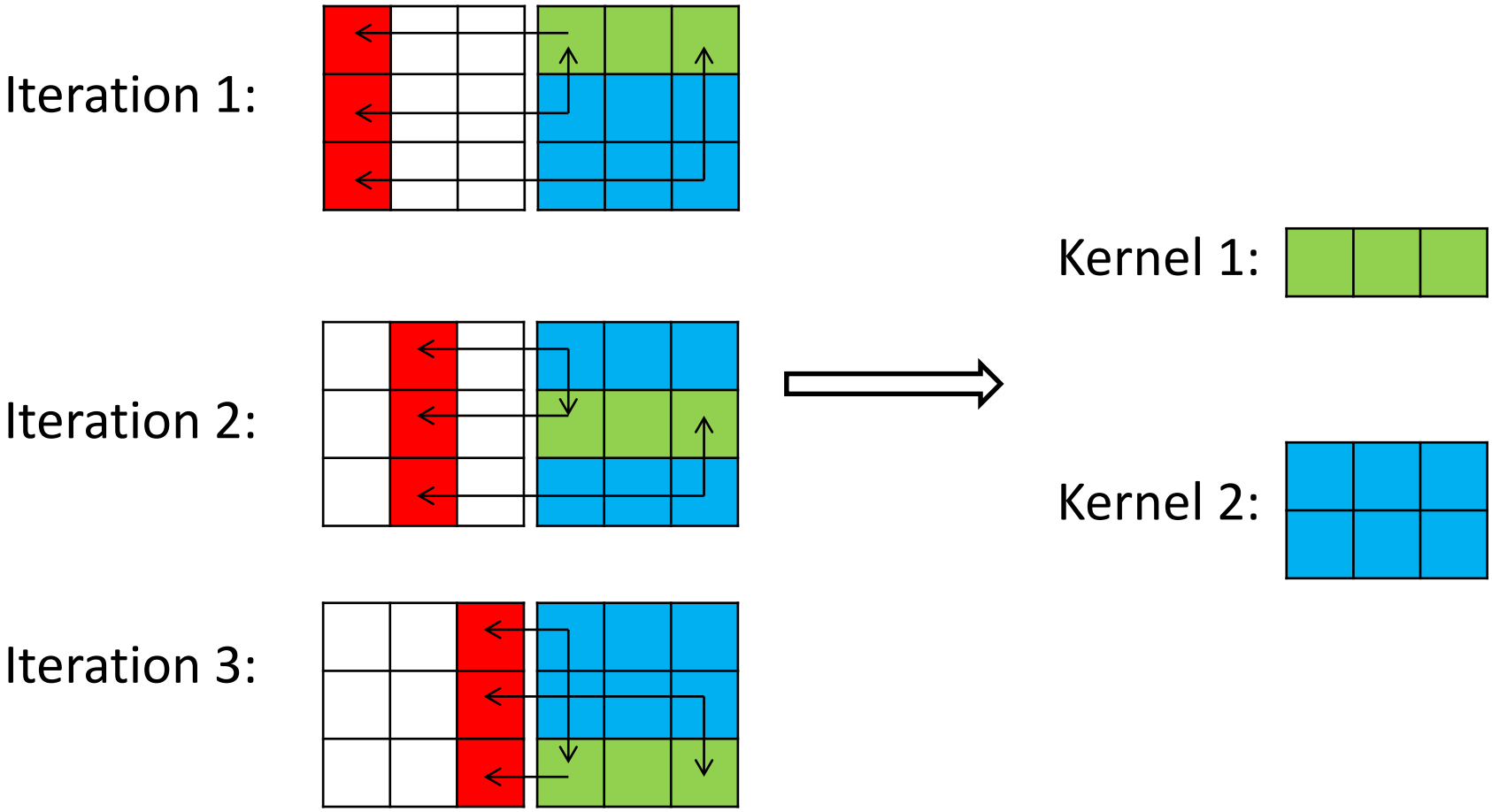
Phase 2

- Two blocks are loaded into a GPU.
- Both blocks are further divided into sub blocks.
- The calculation of sub blocks are not all independent.
- Two kernels are created for calculation.

Multiple GPUs in a single node(cont)



Multiple GPUs in a single node(cont)

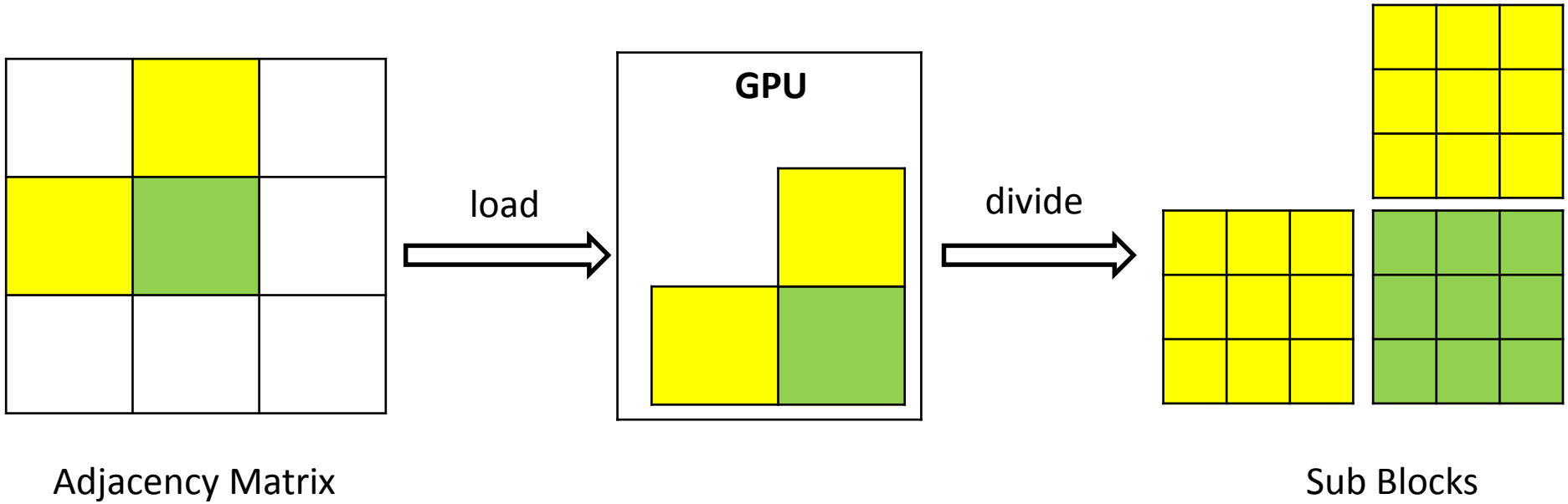


Multiple GPUs in a single node(cont)

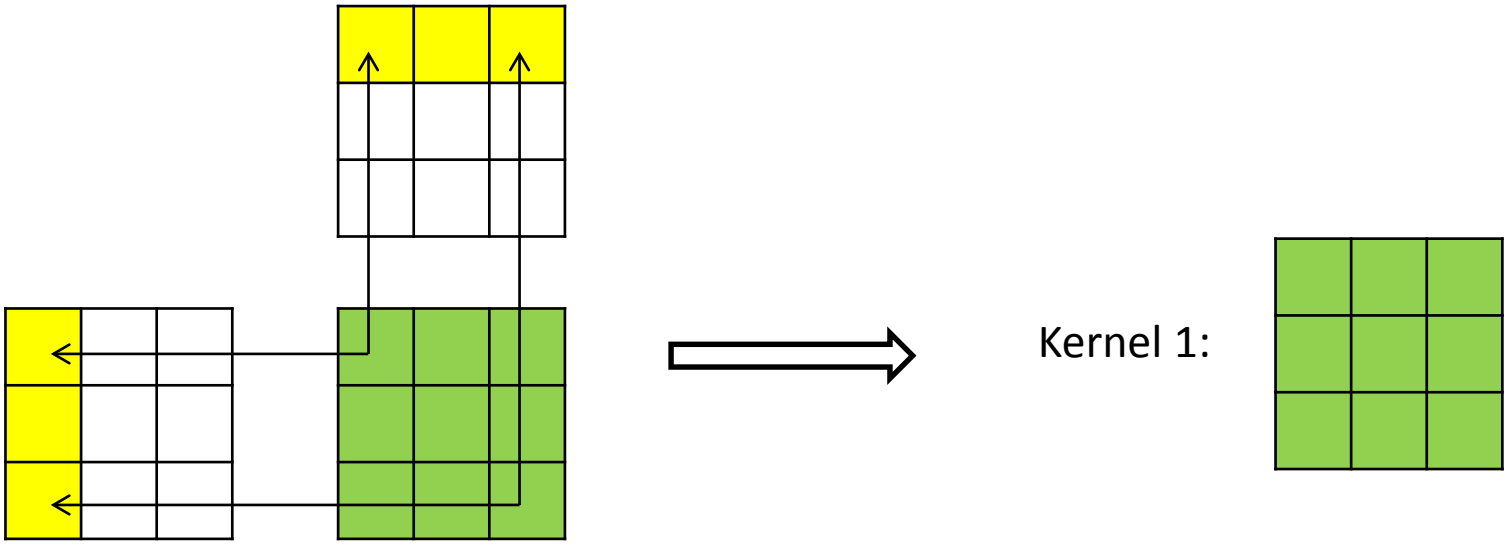
Phase 3

- Three blocks are loaded into GPU.
- All blocks are further divided into sub blocks.
- The calculation of sub blocks are all independent.
- Only one kernel is created for calculation.

Multiple GPUs in a single node(cont)



Multiple GPUs in a single node(cont)



Iteration 1

Multiple GPUs in a Cluster

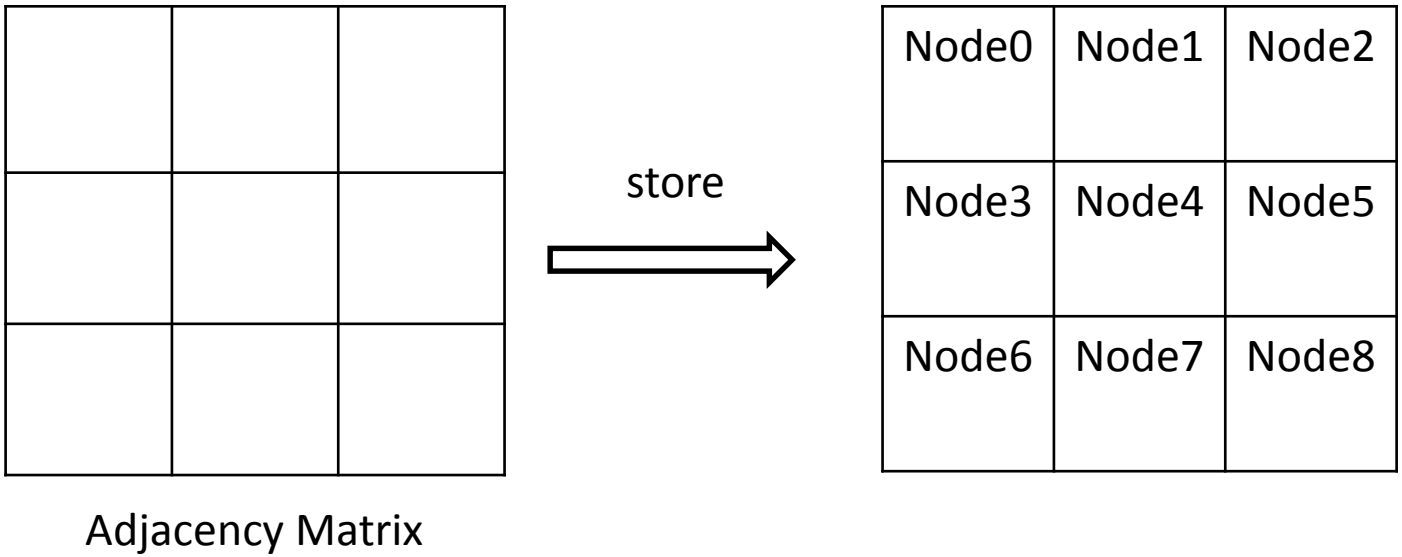
Limitation of Single Node

- Limit number of GPUs.
- Limit size of main memory.

Idea

- Partition adjacency matrix into blocks.
- Several rounds and three phases in a round.
- Each node stores and calculates one active block in certain phase.
- Nodes are communicated between phases.

Multiple GPUs in a Cluster(cont)



Multiple GPUs in a Cluster(cont)

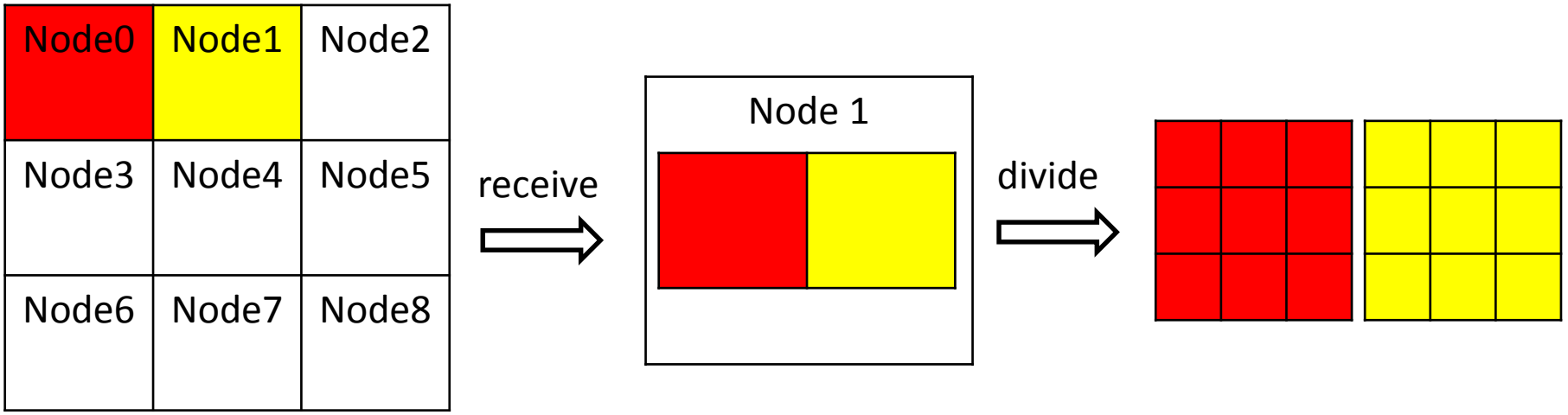
Phase 1

- Only one active node and self dependent.
- Run previous algorithm on the single node.
- After calculation, send the block to active nodes in phase 2.

Phase 2

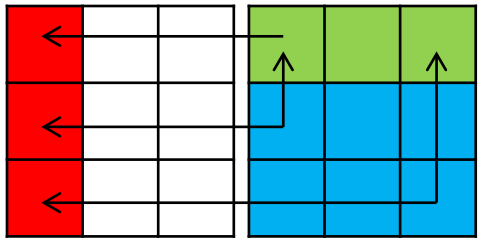
- Active nodes in phase 2 receive block sent from phase 1.
- Both blocks are divided into sub blocks.
- Each sub block is calculated in a GPU.
- The calculation of sub blocks are not all independent.
- The calculation is divided into two steps.
- After calculation, send the block to corresponding active nodes in phase 3.

Multiple GPUs in a Cluster(cont)

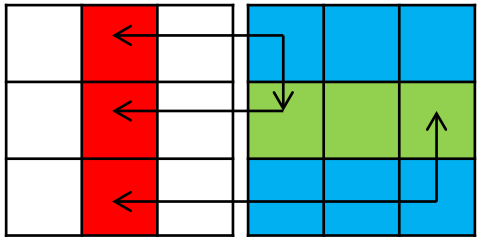


Multiple GPUs in a Cluster(cont)

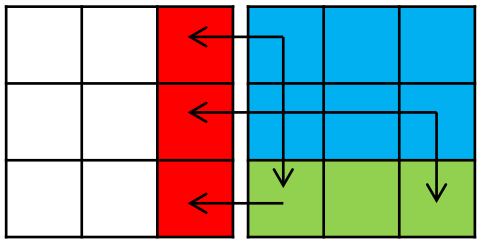
Iteration 1:



Iteration 2:

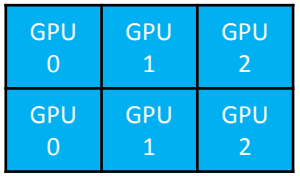


Iteration 3:



Step 1:

Step 2:

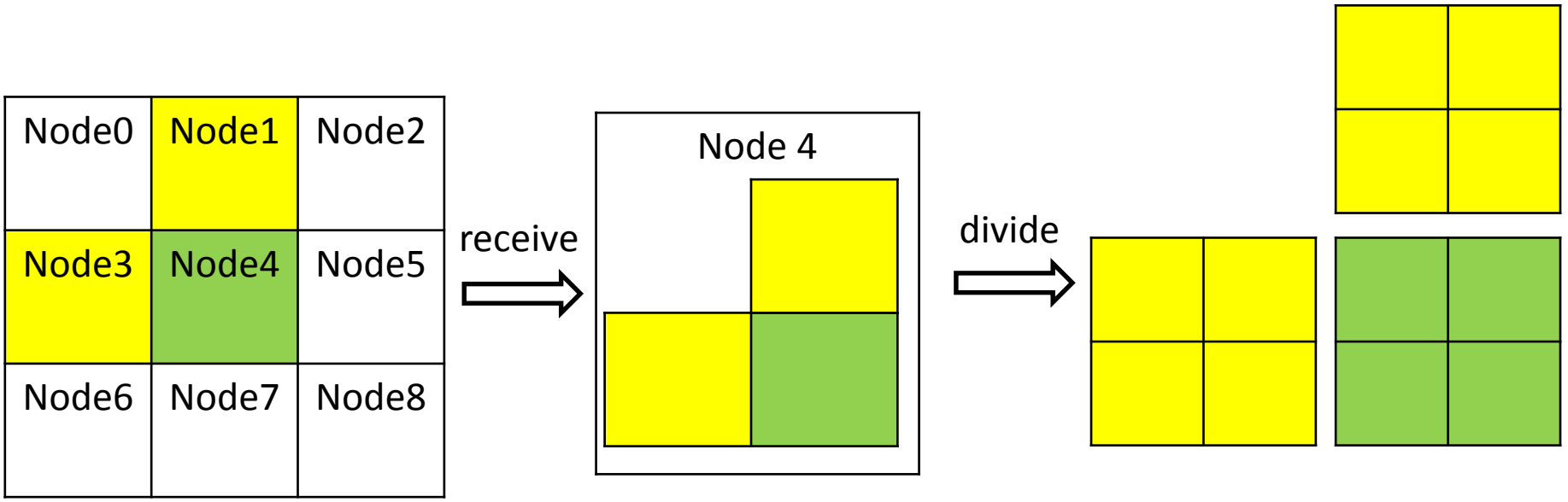


Multiple GPUs in a Cluster(cont)

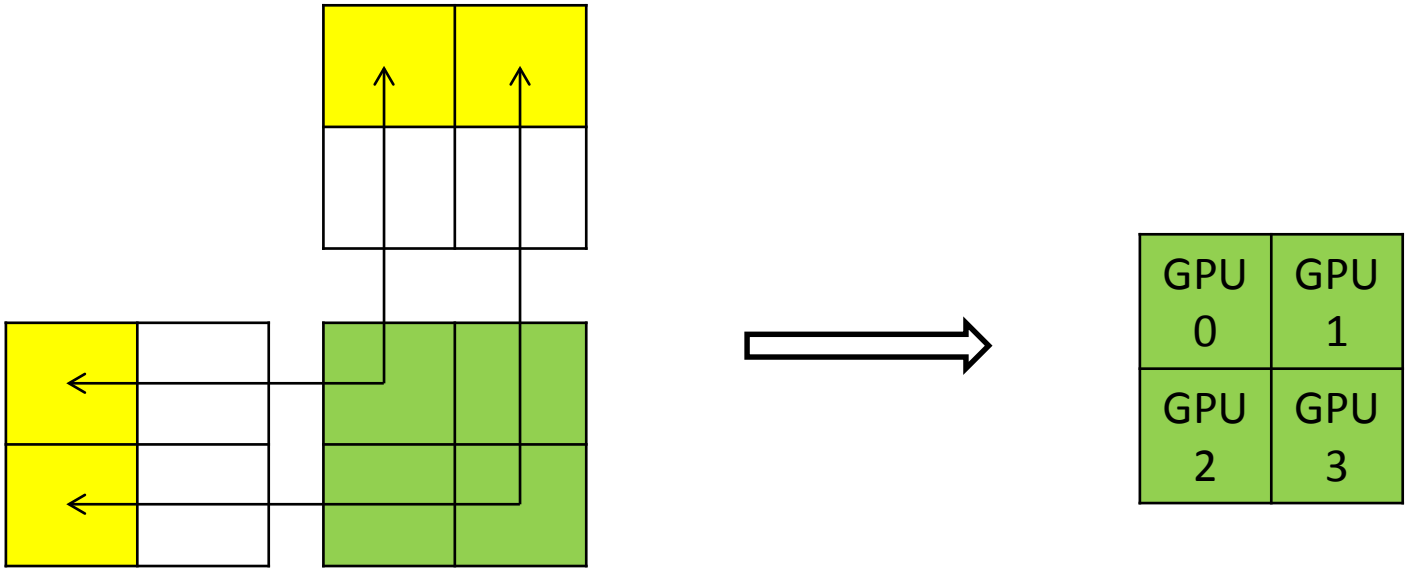
Phase 3

- Active nodes in phase 3 receive two blocks sent from phase 2.
- All blocks are divided into sub blocks.
- Each sub block is calculated in a GPU.
- The calculation of sub blocks are all independent.
- The calculation can be done in one step.

Multiple GPUs in a Cluster(cont)



Multiple GPUs in a Cluster(cont)



Iteration 1

Experimental Setup

Hardware Setup

- A cluster with 11 nodes interconnected with 1 GB/sec ethernet network.
- Each node is equipped with two Intel E5-2650 2 GHz CPUs(16 cores, 32 threads, in total) and four Nvidia Tesla M2090 GPUs and has 64 GB main memory.
- Each M2090 GPU has 512 cores and a 6 GB device memory with the measured peak bandwidth of 120 GB/sec.
- The transfer bandwidth between CPU and GPU in a single node is around 6 GB/sec. The memory bandwidth between GPUs in a single node is nearly 7 GB/sec.

Experimental Setup(cont)

Datasets

- Random graphs
- Scale-free graphs
- R-Mat graphs
- Two real world graphs

Performance of Floyd-Warshall Algorithms in CPU

Algorithms

- Sequential Floyd-Warshall algorithm(FW_S).
- Sequential blocked Floyd-Warshall algorithm(BFW_S).
- Parallel Floyd-Warshall algorithm using OPENMP with 8 threads(FW_P).
- Parallel blocked Floyd-Warshall algorithm using OPENMP with 8 threads(BFW_P).

Datasets

- Three random graphs, three scale-free graphs, three R-Mat graphs and two real graphs.
- Vertex number from 1K to 4K.
- Average degree from 10 to 100.

Performance of Floyd-Warshall Algorithms in CPU(cont)

Random Graph

Vertex Number	Average Degree	FW_S	BFW_S	FW_P	BFW_P
1K	10	8.35	5.22	1.60	1.17
1K	50	8.29	5.24	1.63	1.18
1K	100	8.31	5.18	1.61	1.17
2K	10	66.92	37.18	12.72	7.38
2K	50	66.88	37.21	12.68	7.41
2K	100	66.89	37.24	12.71	7.38
4K	10	531.04	312.38	101.38	61.22
4K	50	530.81	312.56	100.89	61.31
4K	100	531.24	312.72	100.97	61.43

Performance of Floyd-Warshall Algorithms in CPU(cont)

Scale-free Graph

Vertex Number	Average Degree	FW_S	BFW_S	FW_P	BFW_P
1K	10	8.33	5.23	1.61	1.18
1K	50	8.30	5.25	1.66	1.18
1K	100	8.31	5.17	1.62	1.17
2K	10	66.95	37.22	12.72	7.36
2K	50	66.84	37.25	12.66	7.40
2K	100	66.88	37.24	12.75	7.39
4K	10	531.01	312.48	101.40	61.25
4K	50	530.96	312.76	100.86	61.31
4K	100	531.18	312.24	100.91	61.42

Performance of Floyd-Warshall Algorithms in CPU(cont)

R-Mat Graph

Vertex Number	Average Degree	FW_S	BFW_S	FW_P	BFW_P
1K	10	8.25	5.25	1.61	1.17
1K	50	8.28	5.21	1.62	1.18
1K	100	8.32	5.24	1.62	1.18
2K	10	66.90	37.20	12.71	7.42
2K	50	66.79	37.16	12.65	7.40
2K	100	66.85	37.22	12.69	7.45
4K	10	531.32	312.54	101.21	61.21
4K	50	530.79	312.58	100.93	61.29
4K	100	531.11	312.66	100.99	61.45

Performance of Floyd-Warshall Algorithms in CPU(cont)

Real World Graph

Graph	Vertex Number	Average Degree	FW_S	BFW_S	FW_P	BFW_P
ego-Facebook	4K	43.7	530.26	311.99	100.82	60.87
wiki-Vote	7K	29.2	2826.35	1570.18	557.67	311.30

Performance of Floyd-Warshall Algorithms in CPU(cont)

- Performance of Floyd-Warshall algorithm is irrelevant to the edge distribution of graphs. It only depends on the vertex number.

- For simplicity, we ignore the degree distribution of graphs in the following evaluations.

Performance of Katz and Kider's Algorithm

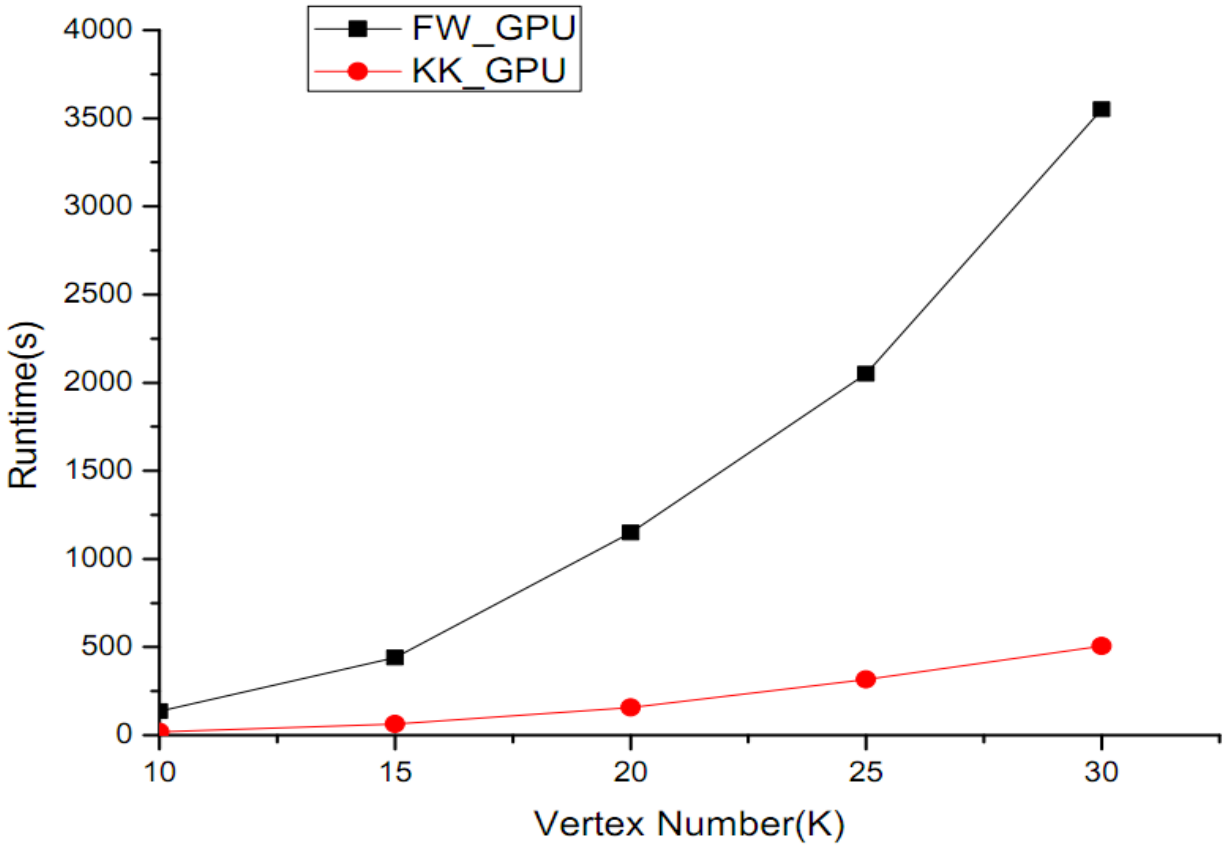
Algorithms

- Floyd-Warshall algorithm on a single GPU(FW_GPU)
- Katz and Kider’s algorithm(KK_GPU)

Datasets

- Graphs with vertex number ranging from 10K to 30K.

Performance of Katz and Kider's Algorithm(cont)



Performance of Extended Algorithm in a Single Node

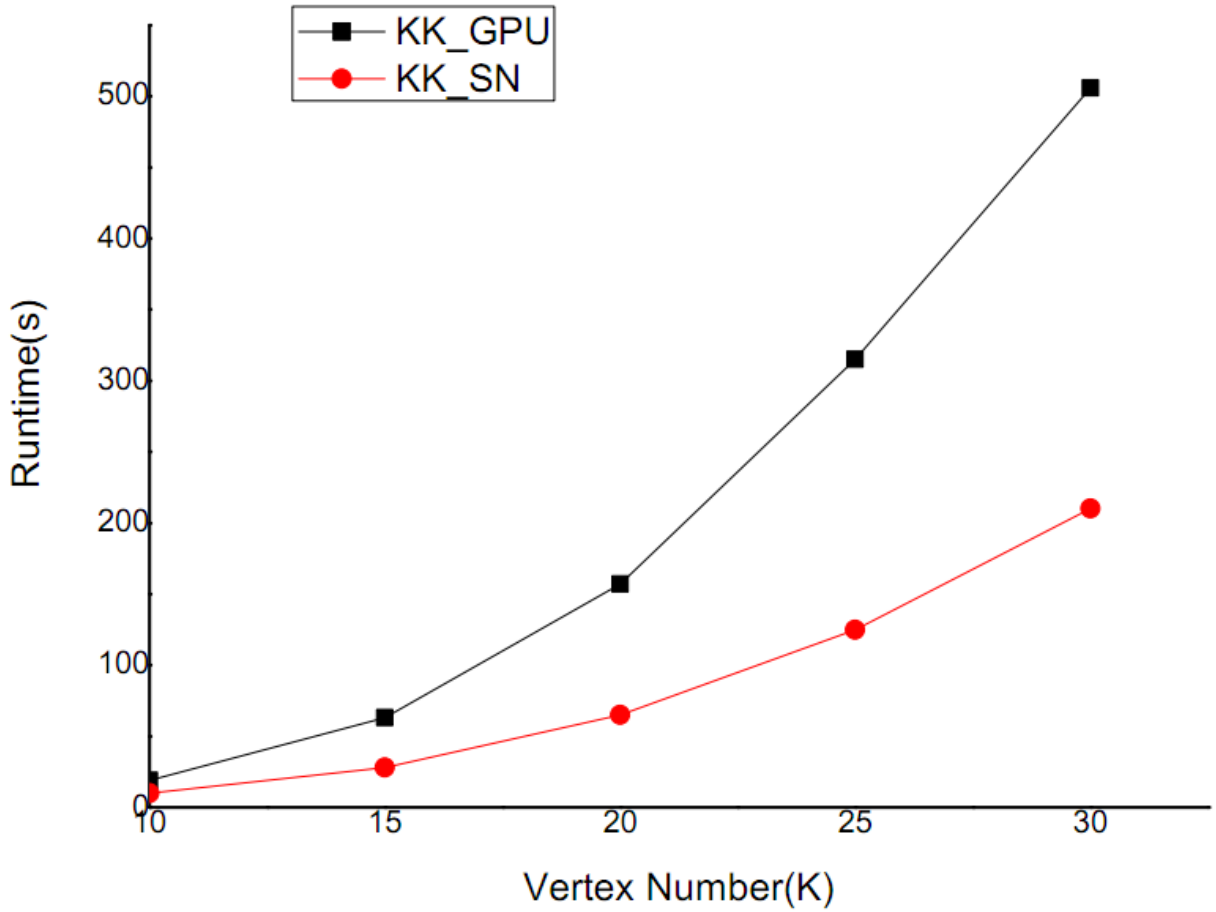
Algorithms

- Katz and Kider's algorithm(KK_GPU)
- Our extended algorithm in a single node with 4 GPUs(KK_SN)

Datasets

- Graphs with vertex number ranging from 10K to 30K.

Performance of Extended Algorithm in a Single Node(cont)



Performance of Extended Algorithm in a Cluster

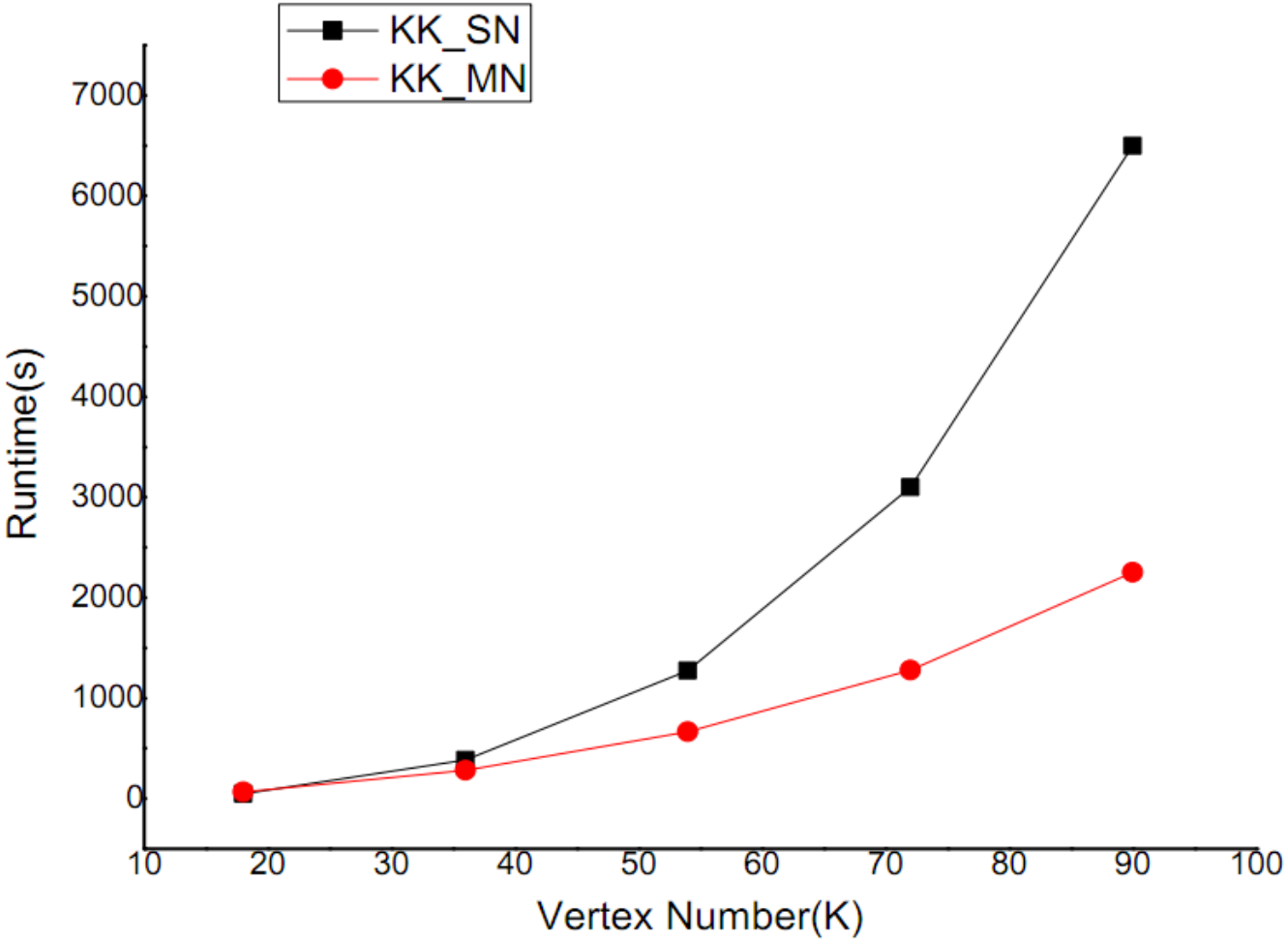
Algorithms

- Our extended algorithm in a single node with 4 GPUs(KK_SN)
- Our extended algorithm run on 9 nodes(KK_MN)

Datasets

- Graphs with vertex number ranging from 18K to 90K.

Performance of Extended Algorithm in a Cluster(cont)



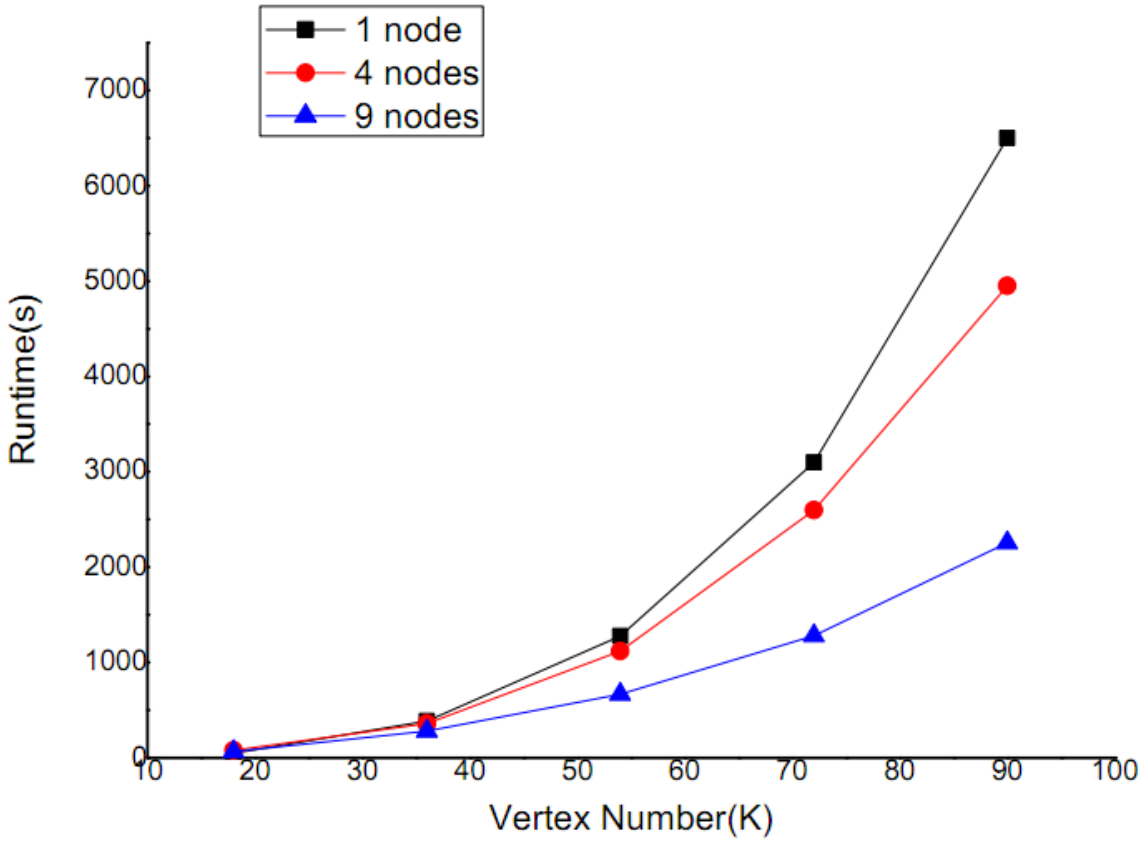
Performance of Extended Algorithm in a Cluster(cont)

Affection of Communication Cost

Vertex Number	Total Time(s)	Communication Time(s)	Ratio
15K	40.66	12.49	30.72%
30K	256.38	49.13	19.12%
45K	808.57	109.71	13.57%
60K	1829.04	190.52	10.42%

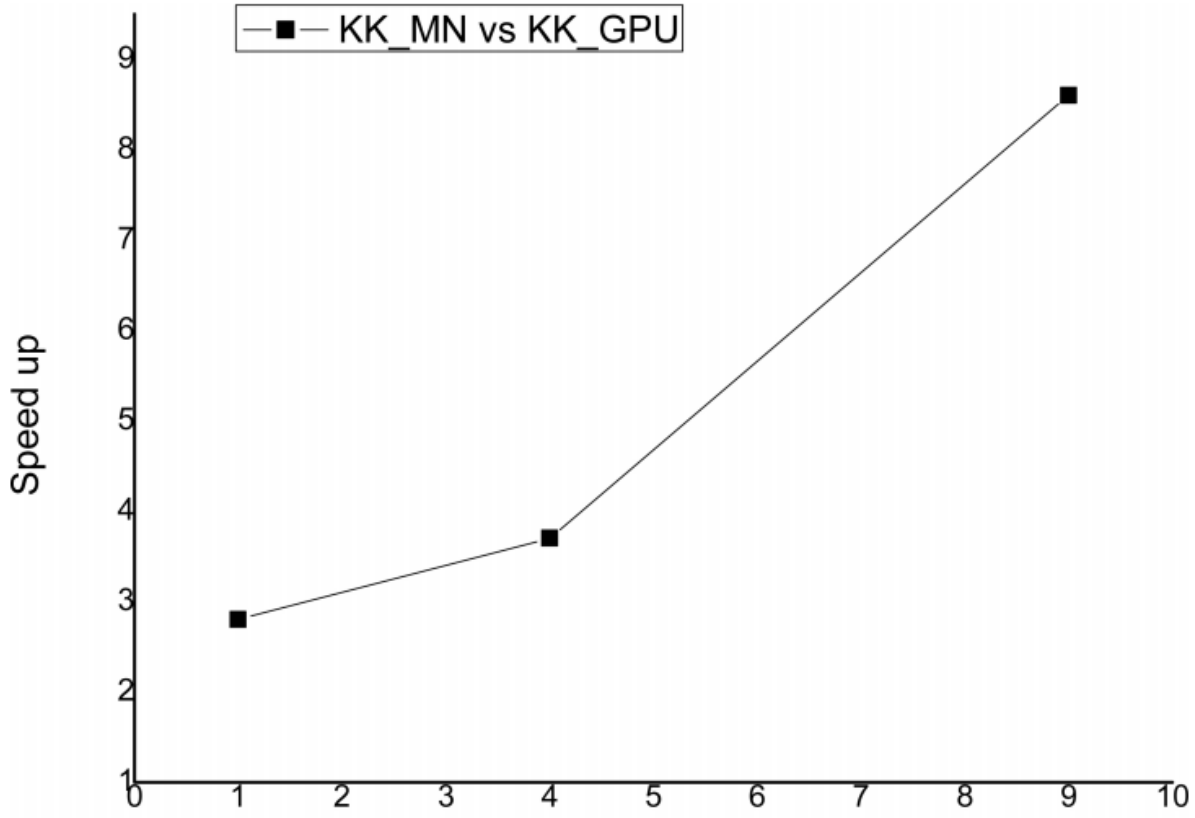
Performance of Extended Algorithm in a Cluster(cont)

Performance of KK_MN under different nodes



Performance of Extended Algorithm in a Cluster(cont)

Speed Up of KK_MN over KK_GPU



Summary

- Our extended algorithms are able to handle large graphs.

- The performance of our extended algorithms scale linearly according to the number of nodes.