Synchronization

Dr. Jiju Poovvancheri
Operating Systems:Recitations
SMU. Winter 2022

https://computing.llnl.gov/tutorials/pthreads/

Pthread mutex (pthread.h)

```
pthread_mutex_t mutex;


/* create and initialize  the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);


//critical section


/* release the mutex lock */
pthread_mutex_unlock(&mutex);

/* Destroy  the mutex lock */
pthread_mutex_destroy(&mutex);
```

POSIX Semaphores (semaphore.h)

sem_init-Initializes a semaphore with a value. Returns 0 if successful, -1 otherwise.

Syntax:
int sem_init(sem_t *sem, int pshared, unsigned int value);

sem-semaphore object to be initialized

pshared-flag indicating whether sem to be shared

value-value to be initialized to sem

e.g.,

```
sem_t sem;
sem_init(&sem, 0, 1);
```

POSIX Semaphores (semaphore.h)

sem_wait- Locks a semaphore and returns 0. If the semaphore value is zero 0, the calling process gets blocked. Returns -1 if unsuccessful (deadlock, interrupt etc..)

Syntax:

```
int sem_wait(sem_t *sem);
```

e.g.,

```
sem_t sem;
sem_wait(&sem);
```

This is the **down(s)** operation

POSIX Semaphores

sem_post- It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

Syntax:

```
int sem_post(sem_t *sem);
```

e.g.,

```
sem_t sem;
sem_post(&sem);
```

This is the **up(s)** operation

POSIX Semaphores

sem_destroy- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Syntax:

```
int sem_destroy(sem_t *sem);
```

e.g.,

```
sem_t sem;
sem_destroy(&sem);
```

Semaphore Vs Mutex?

- Signalling Vs Locking mechanisms
- Mutex lock can be unlocked only by the thread that acquired it. Semaphore value is changed by any thread acquiring or releasing it.
- Single Vs multiple threads

Project 2

Floyd Warshall All Pair Shortest Path

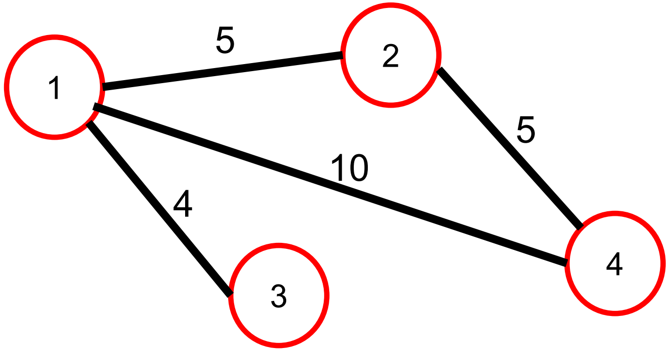
- Given a weighted graph, we want to know the shortest path from one vertex in the graph to another.
  - The Floyd-Warshall algorithm determines the shortest path between all pairs of vertices in a graph.

FWA-Data Structures

- Adjacency matrix (graph)-1 if nodes i and j are connected, else 0.
- Distance matrix (dist)-distance between nodes i and j.

  -dist[i, i]=0;

  -dist[i, j]=INF, if i and j are not connected.



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

graph

$$\begin{pmatrix} 0 & 5 & 4 & 10 \\ 5 & 0 & \text{INF} & 5 \\ 4 & \text{INF} & 0 & \text{INF} \\ 10 & 5 & \text{INF} & 0 \end{pmatrix}$$

dist

FW-Single threaded

Algorithm: Floyd-Warshall (FW) algorithm

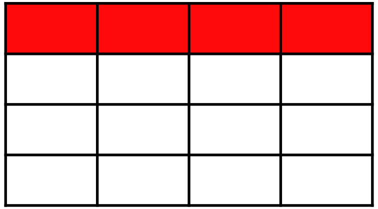
```
for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                        if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j]
                }


        }


}
```

FW-Multi-threaded

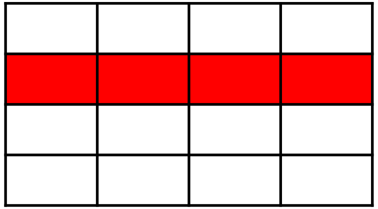
```
for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
                //set up the arguments to be passed, n, k, i
                //create threads → pthread_create


        }
        for (int i = 0; i < n; ++i) {
                //join threads → pthread_join


        }



}
```

Illustration, D^0

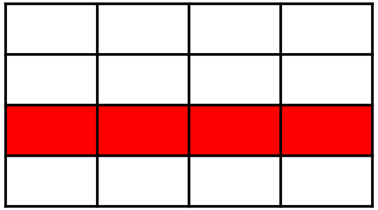
🟥 dist[i, j] 🟦 dist[k, j] 🟩 dist[i, k]



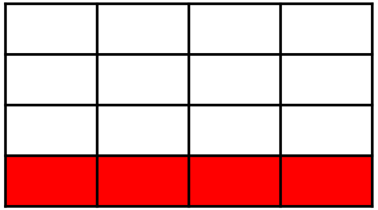
Thread 1



Thread 2

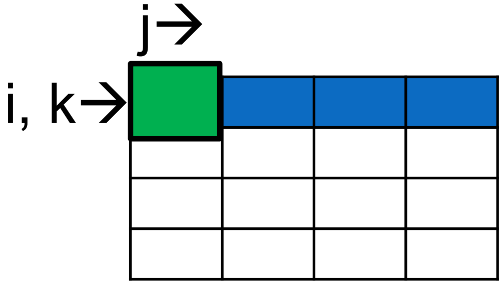


Thread 3

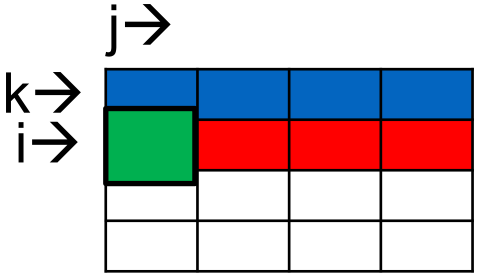


Thread 4

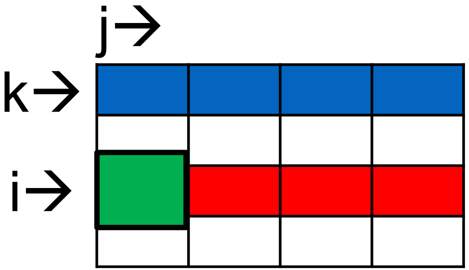
Illustration, D^1 ,k=1



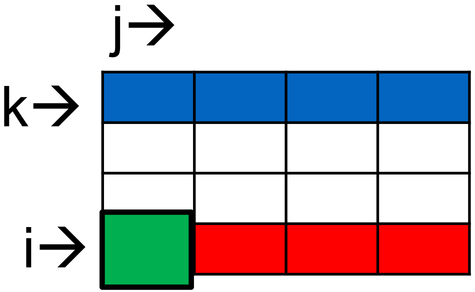
Thread 1



Thread 2

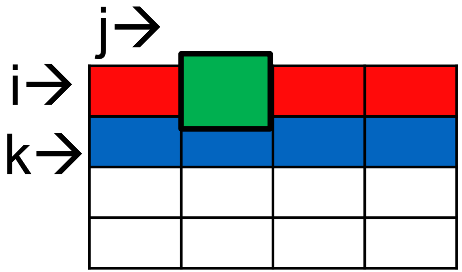


Thread 3

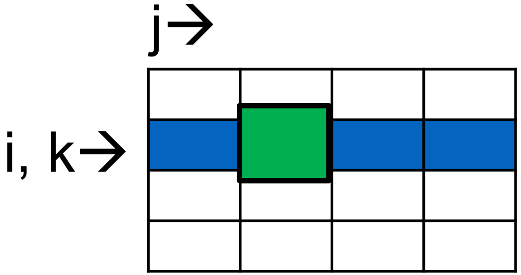


Thread 4

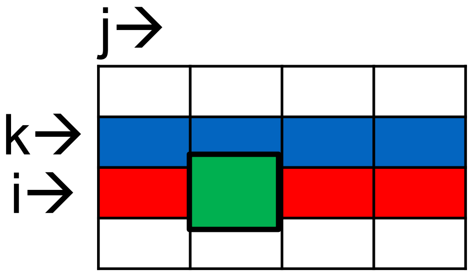
Illustration, D^2,k=2



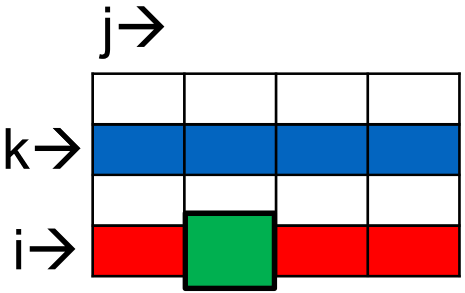
Thread 1



Thread 2

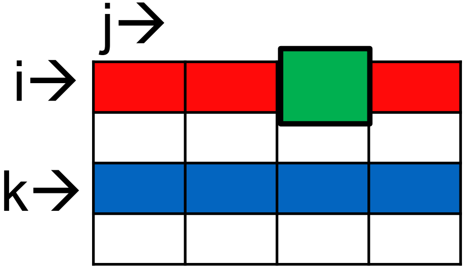


Thread 3

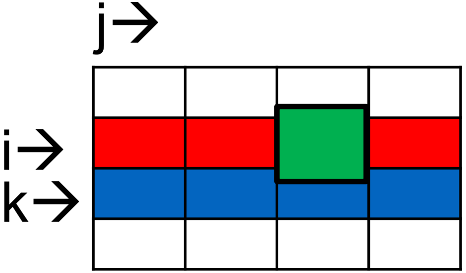


Thread 4

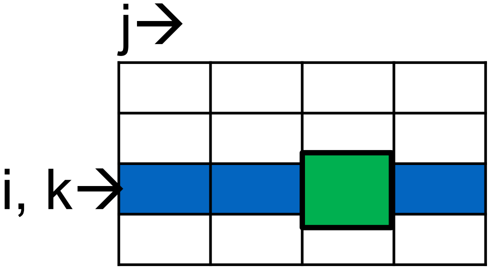
Illustration, D^3 ,k=3



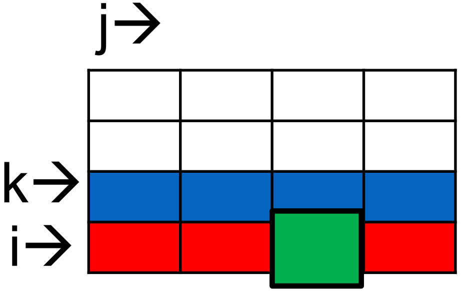
Thread 1



Thread 2

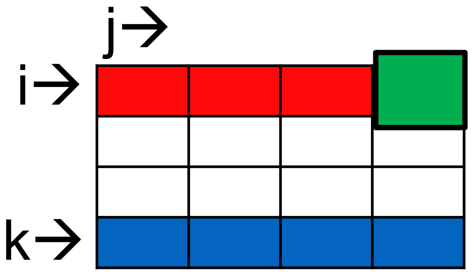


Thread 3

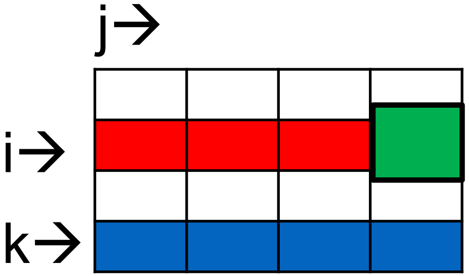


Thread 4

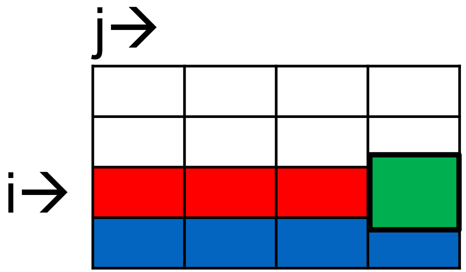
Illustration, D^4 ,k=4



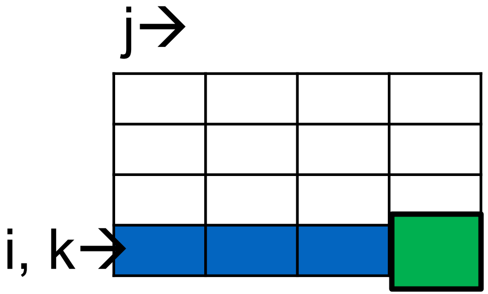
Thread 1



Thread 2



Thread 3



Thread 4

Worker function

```
void *worker(void *args)
{
    //get n,i,k from args
    for (int j = 0; j<n; j++)
    {
        //acquire read lock
        if ((dist[i][k] + dist[k][j]) < dist[i][j])
        {
            //release read lock
            //acquire write lock
            dist[i][j] = dist[i][k] + dist[k][j];
            //release write lock
        }
```

Worker function contd..

```
    else {
              //release read lock
         }

    }
    pthread_exit(NULL);
}
```

Passing arguments

- Use structures to hold values *n*, *k* and *i*

e.g.

```
struct arg_s {
int n;
int i;
int k;
};
```

Passing arguments

- Use structures to hold values *n*, *k* and *i*
- Pass this as arguments when creating pthreads

e.g.

```
int *threads = (pthread_t *)malloc(n *
sizeof(pthread_t));


for (int i = 0; i < n; ++i)
pthread_create((threads+i), NULL, worker, (void
*)&(arguments[i]));
```

Handling large 2D arrays

```
int **graph;
int **dist;
```

global

```
//dynamic mem. allocation for 2D array
graph = malloc(n * sizeof(int *));
for (int i = 0; i < n; ++i) {
     graph[i] = malloc(n * sizeof(int))
}
```