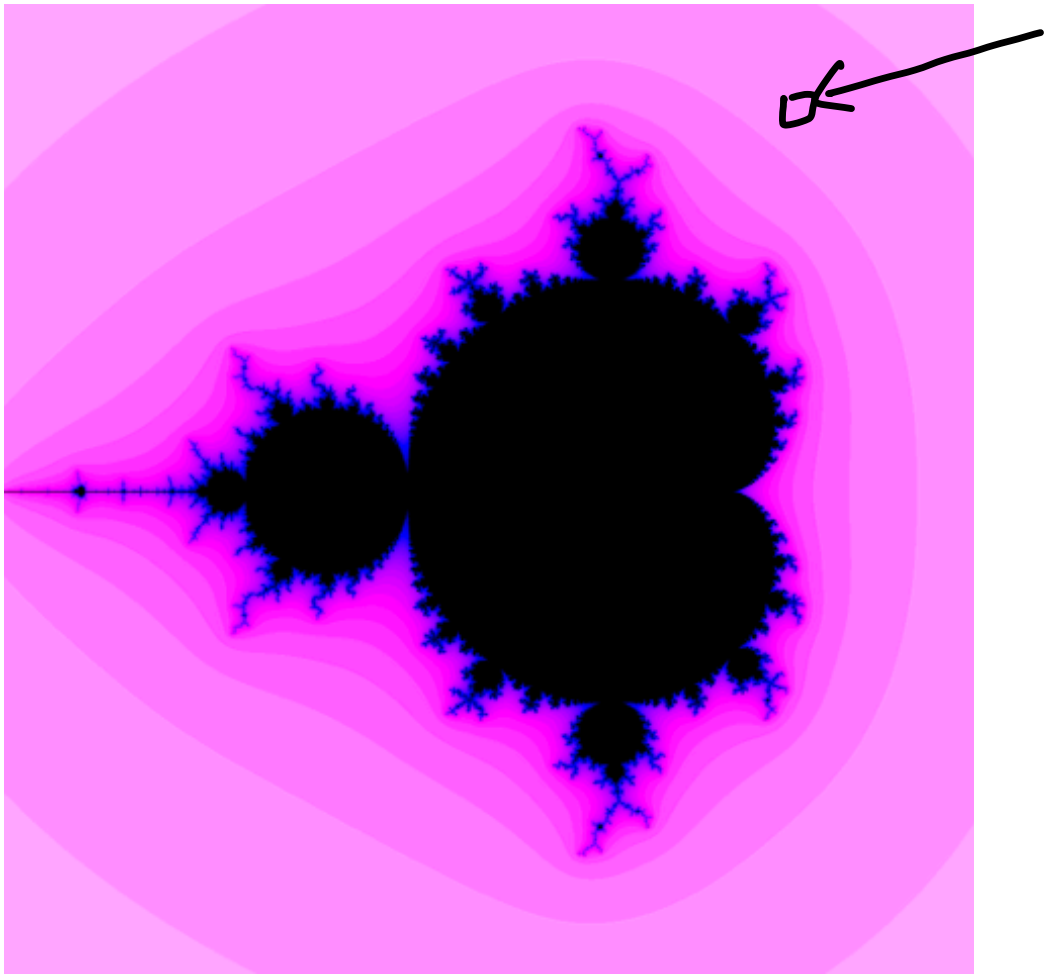
Lecture 16: Thread Pools

COSC 273: Parallel and Distributed Computing

Spring 2023

Mandelbrot Task

Draw this picture as quickly as possible!

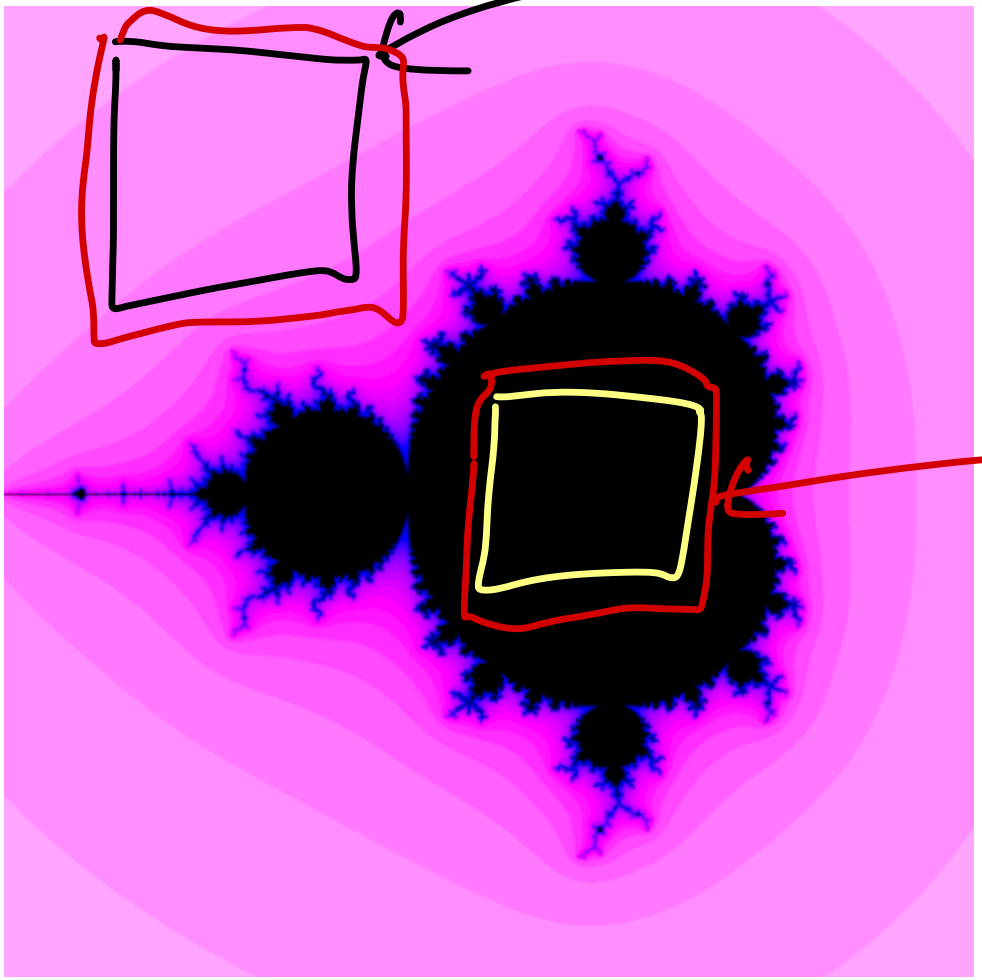


Ideas for Improving Performance?

1. Apply SIMD instructions ← Vector ops
   - perform escape time calculations for multiple pixels at a time
2. Apply multithreading
   - perform calculations for different regions in parallel

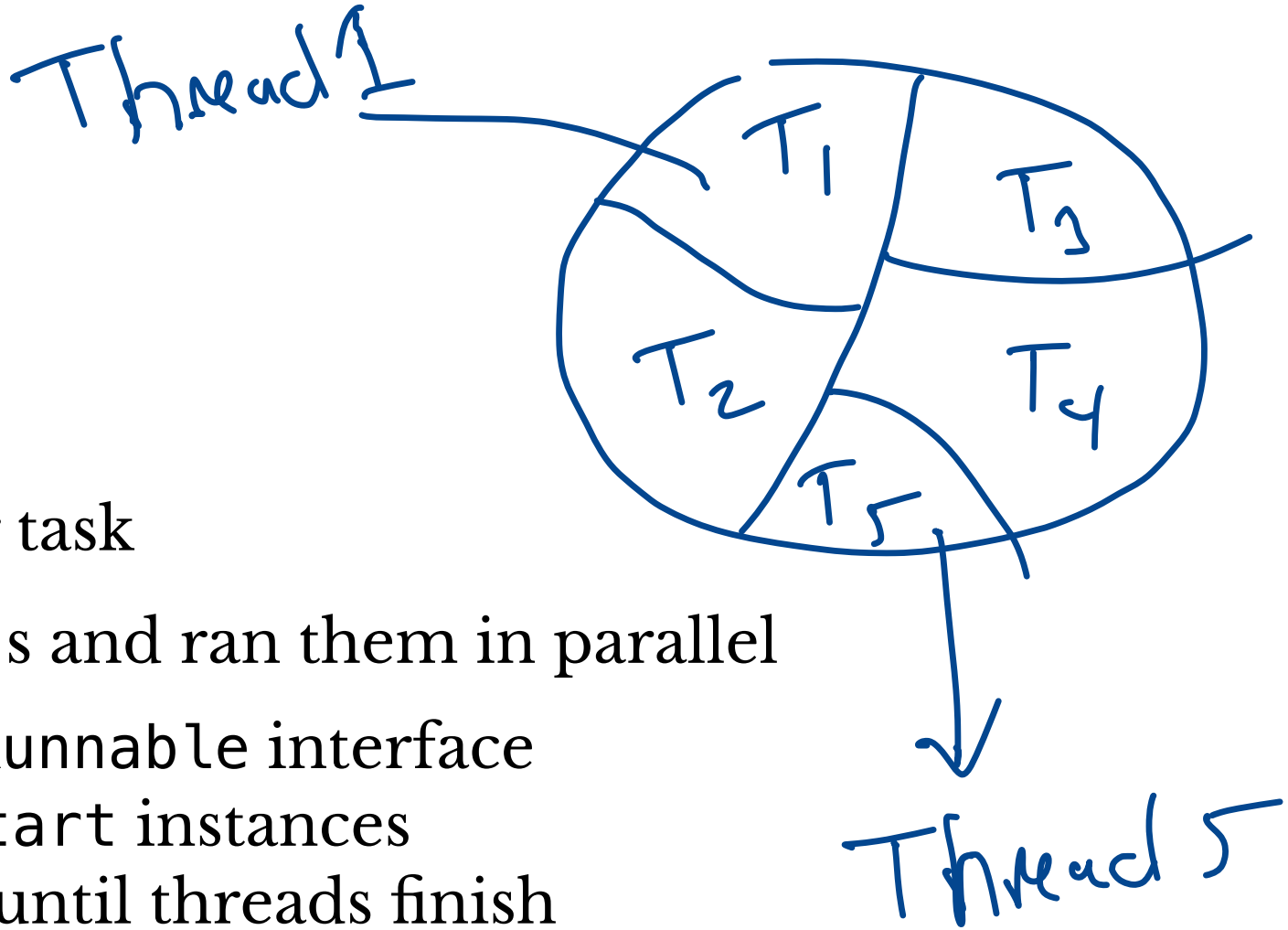
Color = Running Time!

fast running time



slow running

Thread pools



So Far

- One thread per task
- Created `Threads` and ran them in parallel
  - implmenet `Runnable` interface
  - create and `start` instances
  - `join` to wait until threads finish

Example: PiEstimator

```
for (int i = 0; i < numThreads; i++) {
    threads[i] = new Thread(new PiThread(...));
}

for (Thread t : threads) {
    t.start();
}

for (Thread t : threads) {
    try { t.join(); }
    catch (InterruptedException e) { }
}
```

make thread for each task

start all

wait for all to complete

PiEstimator Performance

n threads	pi estimate	time (ms)
1	3.14158	8174
2	3.14161	4690
4	3.14161	2709
8	3.14163	1735
16	3.14156	1867
32	3.14167	1938
64	3.14156	1905
128	3.14157	1907
256	3.14164	1919

# of parallel processors on my computer

10% slower

overhead from managing threads

Observation

Best performance when number of threads = number of available processors

Reasons:

1. Overhead for creating/starting/waiting for threads
2. All tasks require (roughly) same amount of work

Question. What if tasks are different (unkown) amount of work?

tasks = # processors

⇒ total running time ≈ max

More tasks ⟶ divide up work between processors more evenly

Drawbacks of One-Task-Per-Thread

- Creating new `Threads` has significant overhead
  - best performance by balancing number of threads/processors available
- Need to explicitly partition into relatively few pieces
  - partitioning may be unnatural
  - partition may be unbalanced:
    - don't know in advance how long computations will take

When tasks are fairly homogenous (e.g., computing π, shortcuts) previous approach is good

A (Sometimes) Better Way

A nice Java feature: **thread pools**

- Create a (relatively small) pool of threads
- Assign tasks to the pool
- Available threads process tasks
  - if all threads occupied, tasks stored in a queue
  - as threads are completed, threads in pool are reused

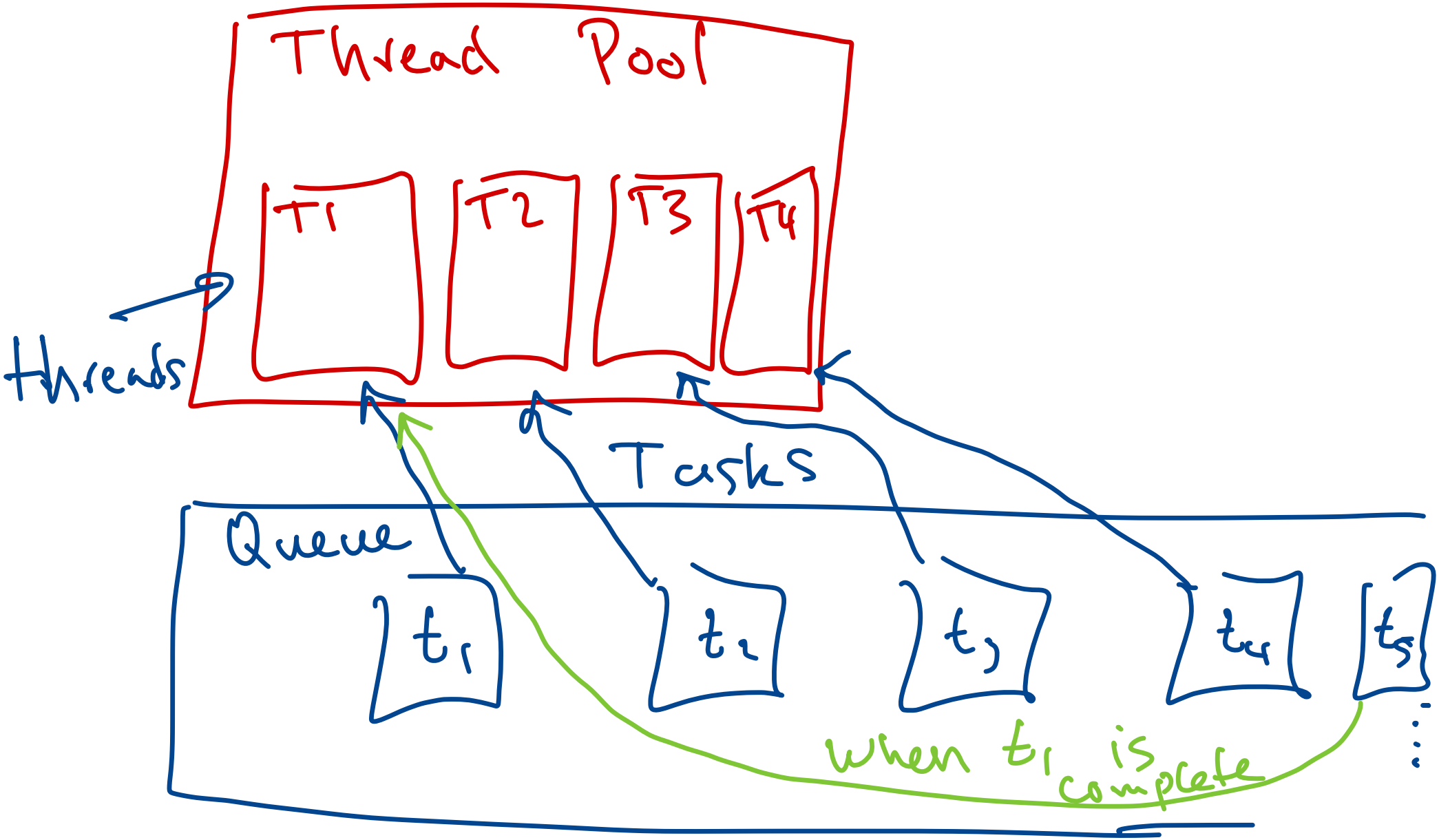
When are Thread Pools Better?

- Many smaller tasks
- Fixed partition of problem may be unbalanced
- “Online” problems: set of tasks not known in advance
  - e.g., processing requests for web server

Thread Pools in Java

- Implement `Executor` interface
  - `void execute(Runnable command)` method
- More control of task handling: `ExecutorService` interface:
  - submit tasks
  - wait for tasks to complete
  - shut down pool (don't accept new tasks)

Thread Pool Picture



Built-in `ExecutorService` Implementations

From `java.util.concurrent.Executors`:

- `newFixedThreadPool(int nThreads)`
  - make a pool with a fixed number of threads
- `newSingleThreadExecutor()`
  - make a pool with a single thread
- `newCachedThreadPool()`
  - make pool that creates new threads as needed (reuses old if available)
- …

Using Thread Pools 1

Define tasks

```
public class MyTask implements Runnable {
    ...
    public void run () {
        ...
    }
}
```

Using Thread Pools 2

Create a pool, e.g., fixed thread pool

```
int nThreads = ...;   # threads in pool

ExecutorService pool = Exercutors.newFixedThreadPool(nThreads);
```

Create and execute tasks

```
MyTask task = new MyTask(...);

pool.execute(task);  ← adds task to queue for the pool
```

Using Thread Pools 3

Shutting down the pool

```
pool.shutdown();
```

Wait for all pending processes to complete (like `join()` method)

Max timeout to wait

```
try {

    pool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);

} catch (InterruptedException e) {

    // do nothing

}
```

Example

Size ≈ 1000

Shortcuts from Lab 02:

```
for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
        float min = Float.MAX_VALUE;
            for (int k = 0; k < size; ++k) {
            float x = matrix[i][k]; float y = matrix[k][j];
            float z = x + y;
            if (z < min)
                min = z;
        }
        shortcuts[i][j] = min;
    }
}
```

Small task t_{ij}

How many small tasks?

1M

A *Small* Task

For fixed row `i`, col `j`:

```
float min = Float.MAX_VALUE;
    for (int k = 0; k < size; ++k) {
    float x = matrix[i][k]; float y = matrix[k][j];
    float z = x + y;
    if (z < min)
        min = z;
}
shortcuts[i][j] = min;
```

size 512

Two Approaches

Approach 1:

- Make a separate thread for each task
  - need `size * size` threads ~ 250k threads

Approach 2:

- Make a thread pool and let the pool decide
  - choose pool size from `availableProcessors()`

- opt # threads (8)

- 250k tasks

Demo

- `executer-shortcuts.zip`

Lab 03 Suggestions

Lab will be posted early next week

1. Make a `Runnable` task that uses SIMD parallelism to compute escape times
2. Use a thread pool to manage tasks

Have a Nice Break!