CMOR 421/521:

Load Balancing and Thread Mapping

Date: 2/13/2024

This Week:

~~Tu: Order Dependency and Synchronization~~

Th: Load Balancing, Reductions, and Thread Mapping

Note: I will be available after class today and Thursday to help with code issues.

Tu	Th	
■	■	1
■	■	2
■	■	3
■	■	4
■	■	5
■	🟨	6
		7
		8
		9
		10
		11
		12
		13
		14
		15

Today

- Load balancing with OpenMP’s parallel-for
- OpenMP reductions
- Load balancing in general
- Thread mapping and domain decomposition

Synchronization and Performance

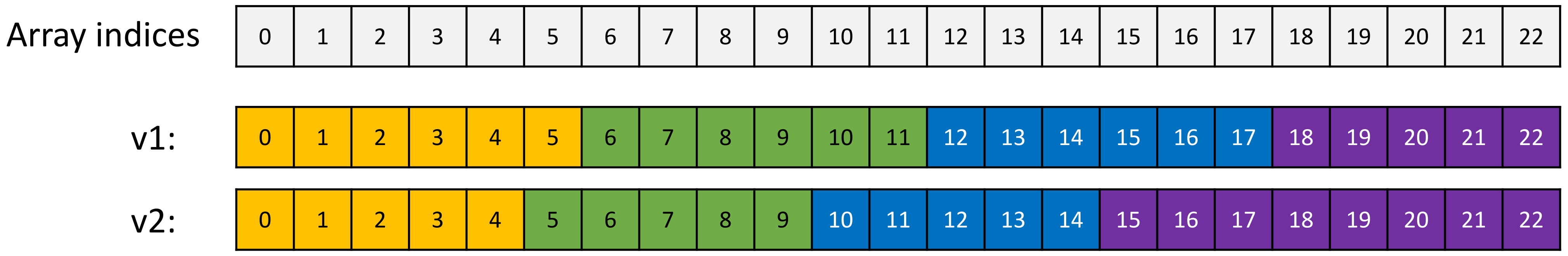
- **Synchronization can cause *deadlocks***
- A deadlock is when one or more threads is unable to proceed
- Can happen with poor use of barriers and other synchronizing directives
  - All threads need to be able to see a barrier
  - **Barriers are NOT unique** and include implicit barriers
    - A thread making it to one barrier can release threads at a different barrier
- Deadlocks will return in MPI **in force**
- **Race conditions and deadlocks are unique to parallel programming – watch out for them!**

Deadlocks and the Fork-Join Model

- **What if the only thread that makes it to the end is thread 0?**
- It’s the only one that was going to proceed anyway
- Might not cause a deadlock!
  - But work may have been uncompleted by the other threads
- **Example code**

Synchronization and Performance

- Synchronization causes idle threads and creates overhead
  - If threads start and finish around the same time, there is less idle time
- Example problem: 23 elements, 4 threads
  - Better to have one thread do more or less than the others?



Synchronization and Performance

- How much time will the different versions take with perfect parallelization?
- i.e., assume all threads:
  - Start at the same time
  - Take the same amount of time to update one element
- How much "work potential" is wasted?
  - v1: (1 idle thread)*(1 step)
  - v2: (3 idle threads)*(3 steps)
- **This is the major motivation behind *load balancing***

v1:

t	0	1	2	3
1	0	6	12	18
2	1	7	13	19
3	2	8	14	20
4	3	9	15	21
5	4	10	16	22
6	5	11	17	

v2:

t	0	1	2	3
1	0	5	10	15
2	1	6	11	16
3	2	7	12	17
4	3	8	13	18
5	4	9	14	19
6				20
7				21
8				22

OpenMP Directives

- **OpenMP tries to provide constructs for common processing patterns**
- One of the most common is for-loops
  - for-loops use often comes with vector/array/set use and sums/products
  - Also have nD arrays/iterations; how to parallelize those?
- How should iterations of a for-loop be spread over the threads?
  - What if it’s a collection of several nested loops?

OpenMP: Parallel for

- **Version 1:** Nested in a parallel block

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < n; i++) {
        A[i] += i;
    } // Note only the word “for” was used
}
```

OpenMP: Parallel for

- **Version 2:** Standalone

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    A[i] += i;
}
// Note now we use “parallel for” and
// the parallel block just uses the for’s
// braces
```

Notes on OpenMP's parallel-for

- <u>**You must use vanilla for-loops**</u>
  - Start at a given value, end at another (unchanging value), **step by 1**
- Each thread works on a contiguous chunk of iterations
  - They do not take num_threads sized steps
- **By default, it tries to distribute an equal number of iterations to each thread**
  - Have other options as well, known as *schedules*

Load Balancing: Scheduling

OpenMP offers some clauses to try to load balance for-loops using different strategies

- static
- dynamic
- guided
- auto
- runtime

- **Usage:** `#pragma omp (parallel) for schedule(<type>)`

Scheduling: static

- Static scheduling will try to assign equal chunks of iterations to each thread
- The chunk size can be specified; if not provided it will be one chunk per thread (at most)

```
#pragma omp parallel for schedule(static, chunk_size)
for(i = 0; i < n; i++){
    do_stuff(i)
}
```

Scheduling: dynamic

- Each thread is given a chunk of the specified size and grabs a new chunk once it’s finished until no chunks are left
- chunk_size doesn’t need to be specified, default is 1

```
#pragma omp parallel for schedule(dynamic, chunk_size)
for(i = 0; i < n; i++){
    do_stuff(i);
}
```

Scheduling: guided

- Like dynamic, each thread works on a chunk then grabs a new chunk once it is finished
- However, the chunk size can vary. The chunk size is *proportional* to the # of remaining itr / num threads
- chunk_size here is the *minimum* chunk_size, default 1

```
#pragma omp parallel for schedule(guided, chunk_size)
for(i = 0; i < n; i++){
    do_stuff(i);
}
```

Scheduling: auto

- The compiler or runtime system gets to pick the scheduling
- Useful when on a shared system

```
#pragma omp parallel for schedule(auto)
for(i = 0; i < n; i++){
    do_stuff(i);
}
```

Scheduling: runtime

- The schedule type will be determined at runtime using the OMP_SCHEDULE environment variable
- Or: omp_set_schedule(<schedule-type>)

```
#pragma omp parallel for schedule(runtime)
for(i = 0; i < n; i++){
    do_stuff(
}
```

Coding Example: parallel-for schedules

OpenMP: Nested Loops

- Sometimes we have nested for-loops though, and the first dimension may not be very large
- Can use the “collapse” clause

```
#pragma omp parallel for collapse(2)
for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         do_stuff(A[i,j]);
    }
}
```

OpenMP Reductions

- OpenMP gives us ways to parallelize common processing patterns
- One example of such a pattern is a ***reduction***
  - Finding the min or max
  - Computing a sum, product, average, etc

OpenMP Reduction

- Reductions are applied to parallel for loops to ***reduce*** answers on each thread to a single answer
- Each thread gets their own private version of the reduction variable to work with in the loop

```
int var;
#pragma omp parallel for reduction(command:var)
{
    // var is now private inside the loop
}
```

OpenMP Reduction Types

Different commands can be used for a reduction:

- +,-
- *
- min
- max
- &&, ||
- &, |, ^
- Use OpenMP’s reduction over your own code whenever possible!
  - It is optimized in ways you likely will not think of

What Can Go Wrong?

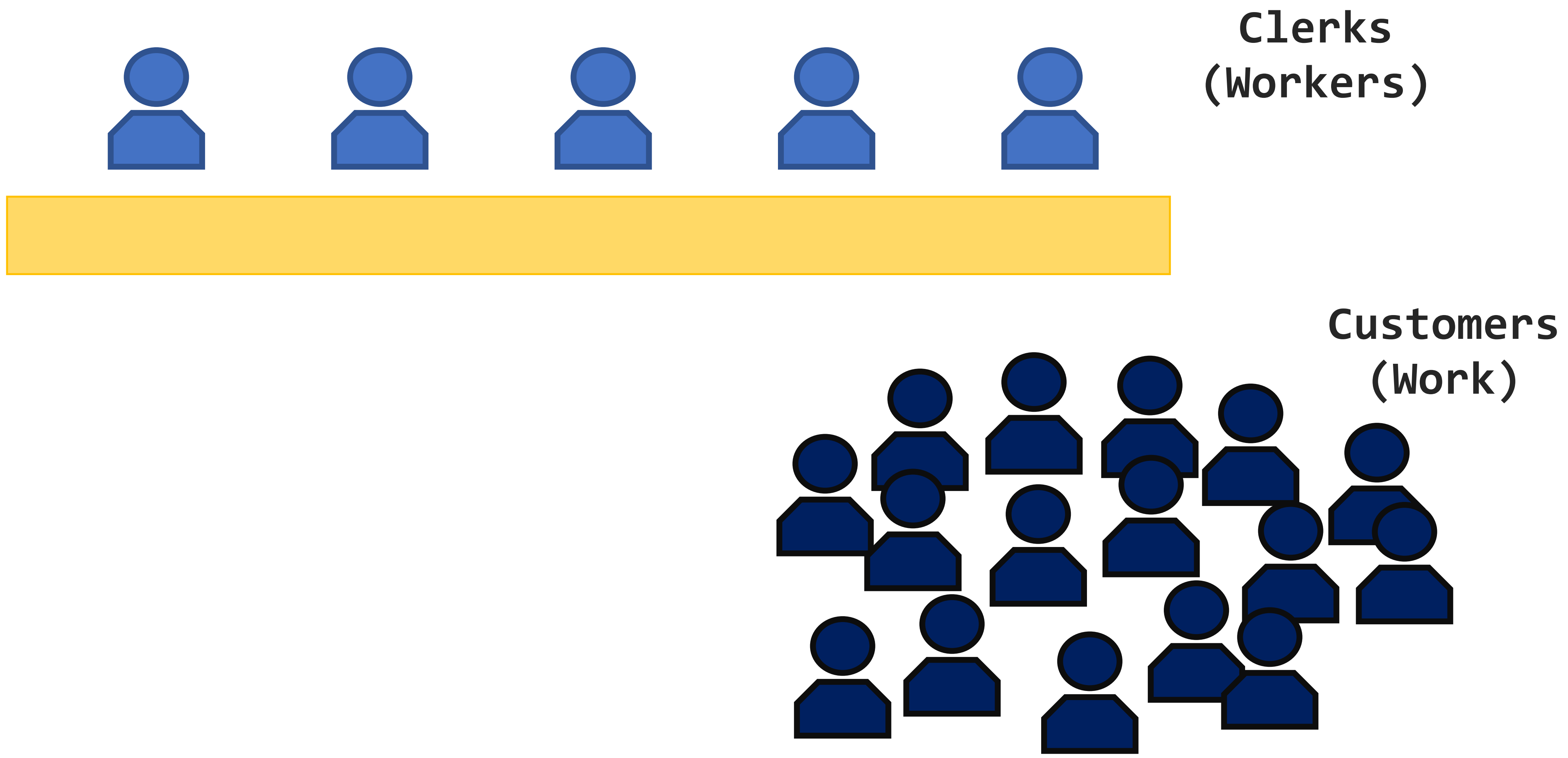
- What if the number of iterations is less than the number of threads?
  - What if they are comparable?
  - It can be the same or slower than a serial code!
    - Parallelism comes with overhead
    - We now have multiple caches being managed
    - Very small codes often do not benefit from parallelism
- **What if the “work” per iteration is not constant?**

Why Load Balance?

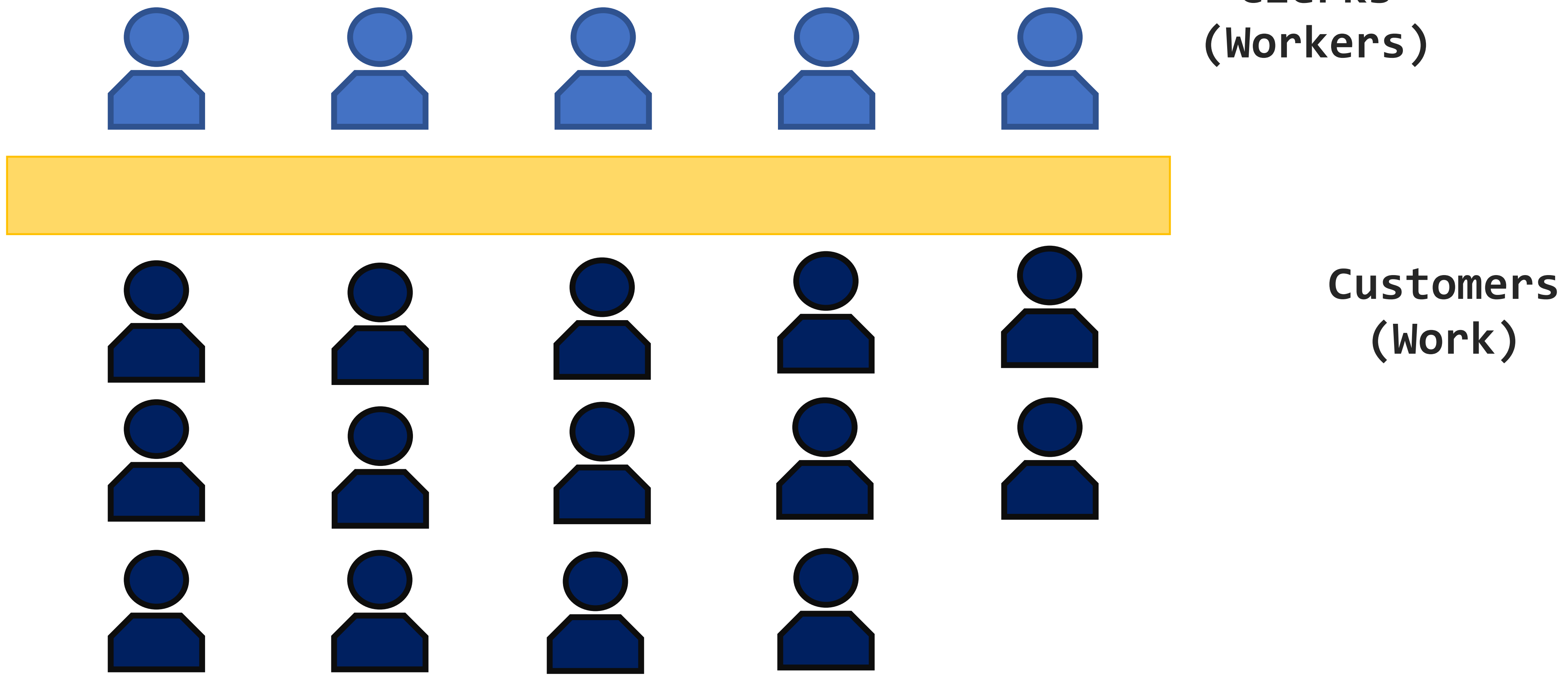
Previously: "A well-used system is never idle"

- Sometimes computing resources are dedicated to your program, you have them whether you use them or not
  - If you are not using them, they are being wasted
- Sometimes you are **sharing** a system, and you have resources until your program finishes
  - If you are not using them, you are wasting other peoples' time too

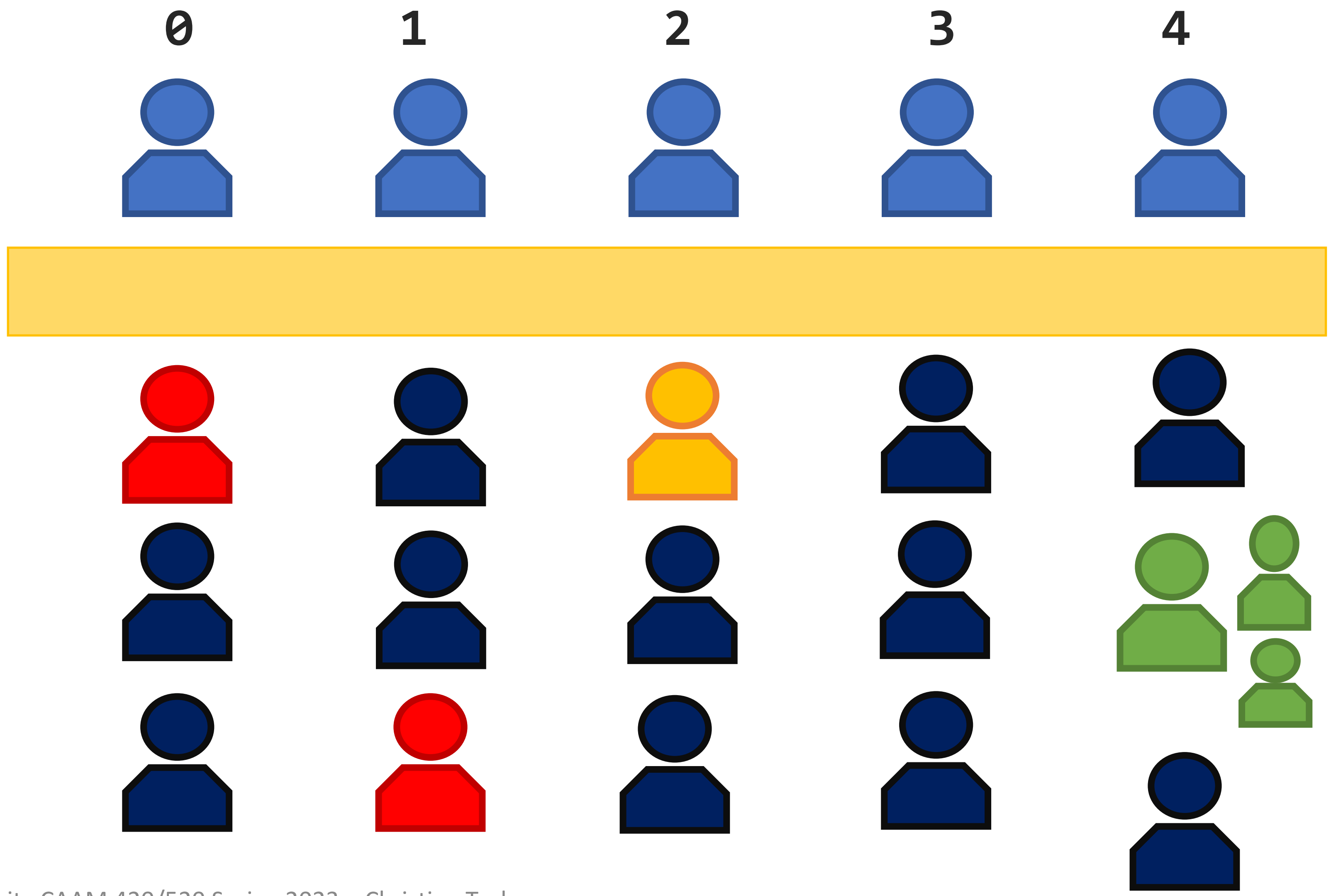
Example: Grocery Checkout



Ideal Situation: Equal Work/Worker



More Realistic Situation

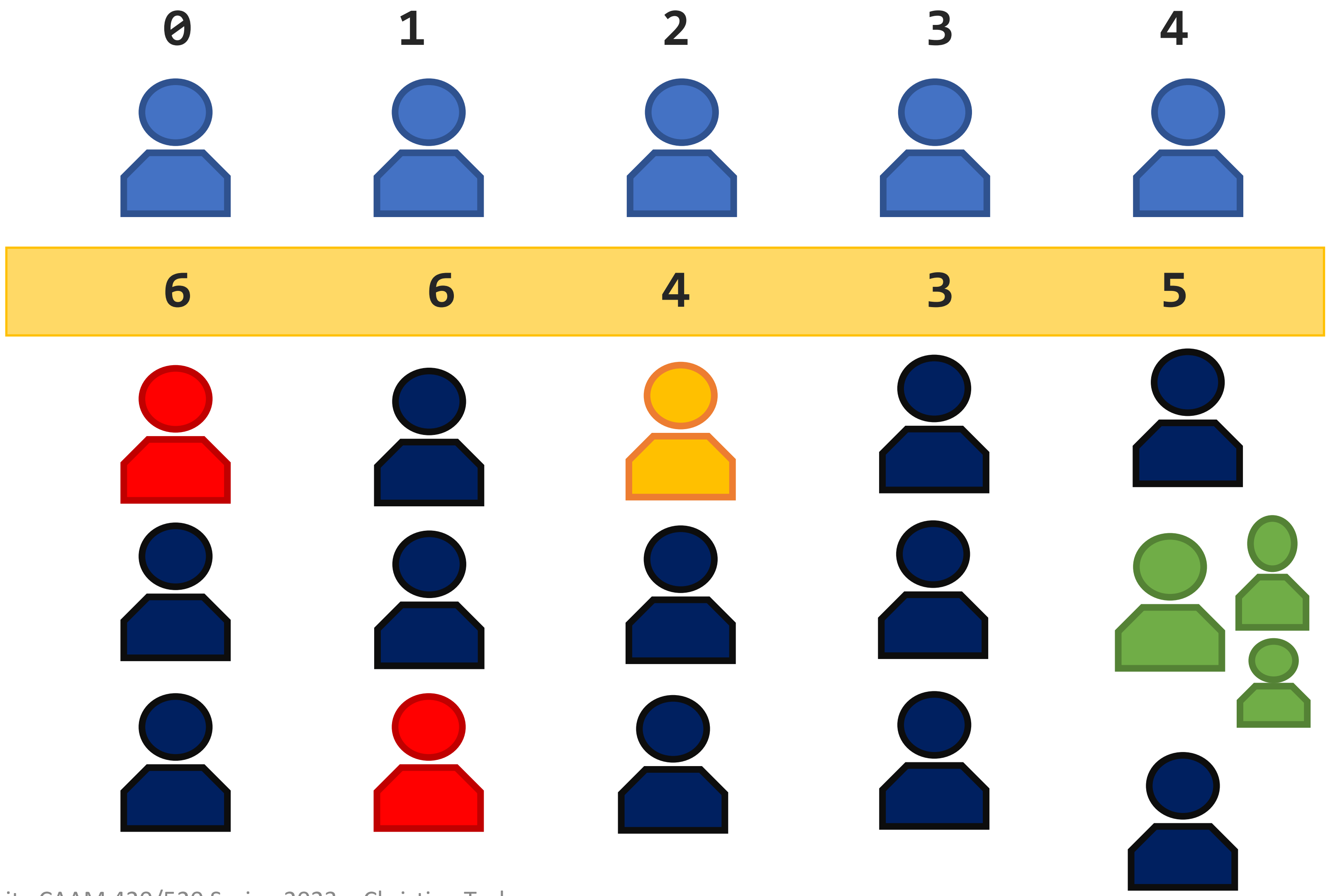


Clerks (Workers)

Work Load

🔴	4
🟠	2
🟢	n x 1
🔵	1

More Realistic Situation

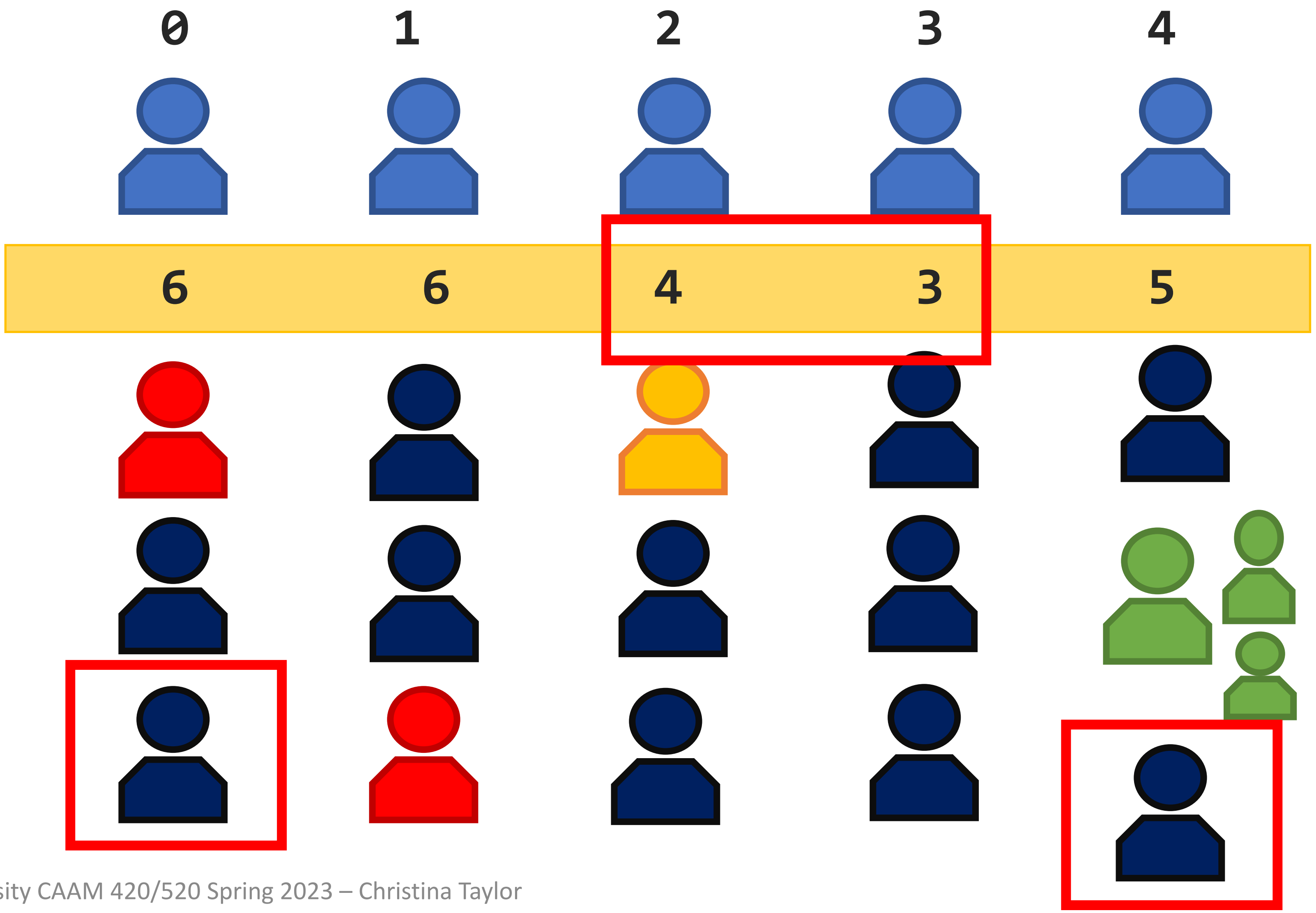


t	0	1	2	3	4
1	1	1	1	1	1
2		2		2	2
3		3	2	3	
4			3		
5	2				1
6	3				

Work Load

🔴	4
🟡	2
🟢	n x 1
🔵	1

More Realistic Situation



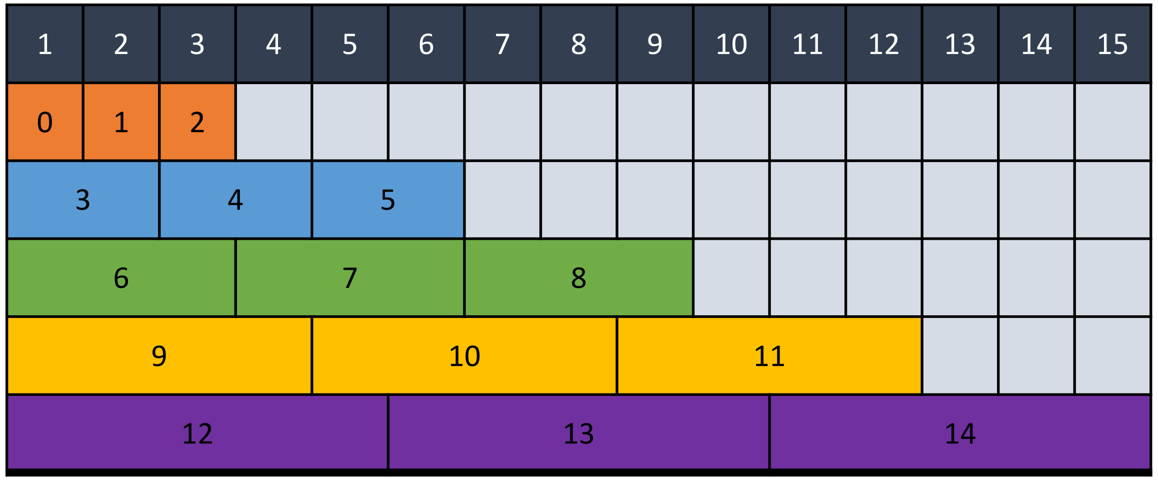
t	0	1	2	3	4
1	🟥 1	1	🟨 1	1	1
2	🟥	2	🟨	2	🟩 2
3	🟥	🟥 3	2	3	🟩
4	🟥	🟥	3		🟩
5	2	🟥			1
6	3	🟥			

t	0	1	2	3	4
1	🟥 1	🟥 1	🟨 1	1	🟩 1
2	🟥	🟥	🟨	2	🟩
3	🟥	🟥	2	3	🟩
4	🟥	🟥	3	4	2
5	2	2	4	5	
6					

Load Balancing w Non-Constant Work

Iteration:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Work:	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5

Default Scheduling: 30 wasted iterations

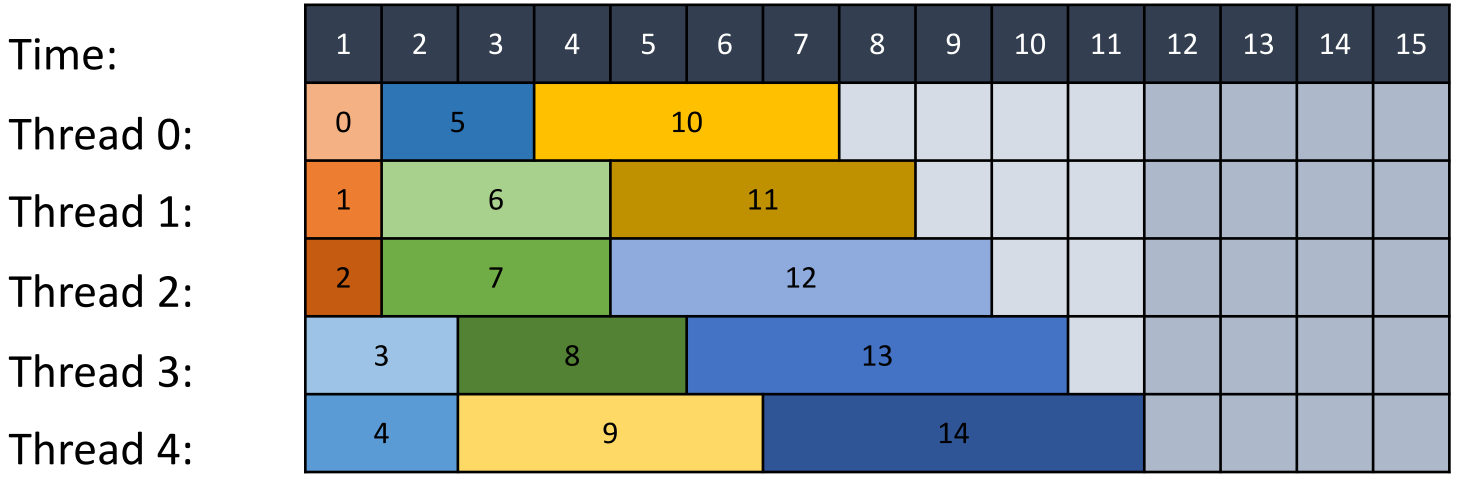


Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Thread 0:	0	1	2												
Thread 1:	3		4		5										
Thread 2:	6			7			8								
Thread 3:	9				10				11						
Thread 4:	12					13					14				

Load Balancing w Non-Constant Work

Iteration:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Work:	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5

Dynamic Scheduling (chunk size = 1): 10 wasted iterations



Code: Scheduling w Non-Constant Work

Load Balancing: “Homespun”

- If your problem is friendly enough, you can figure out how to balance it yourself

Version 1: Bad load balancing

```
// A is a ragged array of n rows
// and m[i] entries per row
#pragma omp parallel for
for(i = 0; i < n; i++) {
    for(j = 0; j < m[i]; j++) {
        printf(“%d\n”, A[i][j]);
    }
}
```

Load Balancing: “Homespun”

- If your problem is friendly enough, you can figure out how to balance it yourself

Version 2: Good load balancing

```
// M = sum(m[i], i=0,…, n-1)
#pragma omp parallel for
for(I = 0; I < M; I++){
    // calc i, j from I
    printf(“%d\n”, A[i][j]);
}
```

Load Balancing: “Homespun”

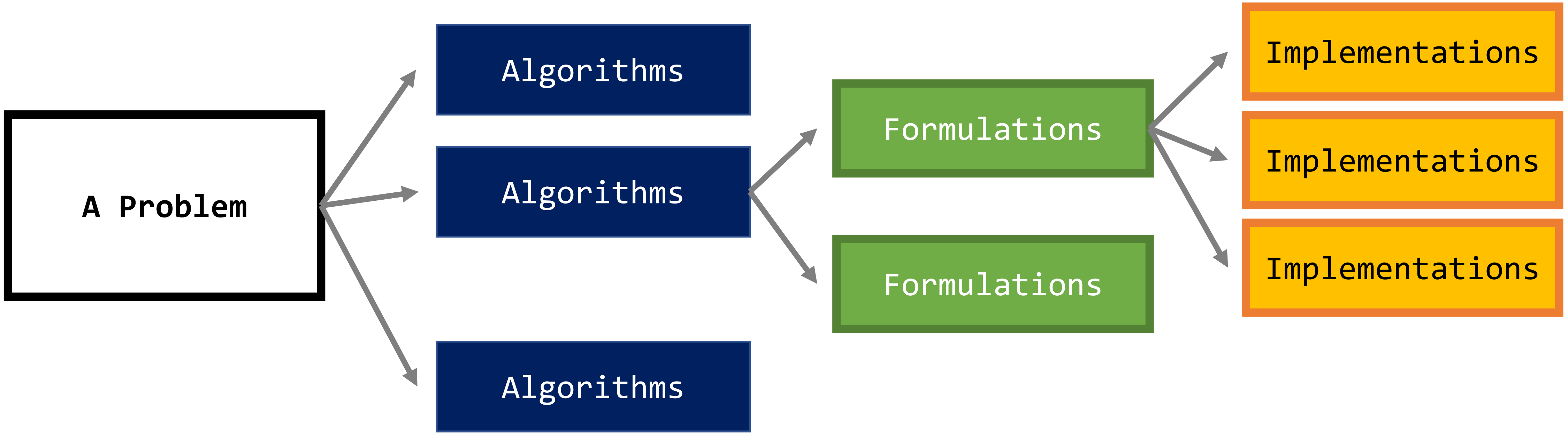
- **Note:** Homespun isn’t always cheap or possible

Version 2: Good load balancing

```
i = 0;
j = 0;
#pragma omp parallel for
for(I = 0; I < N_total; I++){
    // calc i, j from I
    if (I == m[i]) {
        i++;
        j = 0;
    }
    printf(“%d\n”, A[i][j]);
    j++;
}
```

Solving Problems in Parallel

Metrics: Time, Memory, and Parallel Performance



Metrics: Theoretical properties (e.g. error bounds, convergence rates, etc)

Real World Parallelization

- **Sometimes you can’t get all threads active**
  - There may be an order dependency that is too strong to allow full parallelization
- We may be able to identify chunks of the problem that CAN be done in parallel
  - Breaking the problem into chunks of parallelizable work is called ***domain decomposition***
  - Assigning chunks to threads is called ***thread mapping***
- There will/may be a portion of the program where **not all threads are active**
  - These are the called the ***spin-up/spin-down*** phases

Wavefront Parallelization: 2D BVP

Example: 2D Boundary Value Problem

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0, \qquad x > 0, y > 0$$

$$u(x = 0, y) = f(y)$$

$$u(x, y = 0) = g(x)$$

Wavefront Parallelization: FD Formulas

Example: 2D Boundary Value Problem

$$\frac{\partial u}{\partial x} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x}, \qquad \frac{\partial u}{\partial y} = \frac{u_{i,j} - u_{i,j-1}}{\Delta y}$$

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0 \Rightarrow \frac{u_{i,j} - u_{i-1,j}}{\Delta x} + \frac{u_{i,j} - u_{i,j-1}}{\Delta y} = 0$$

Wavefront Parallelization: FD Formulas

Example: 2D Boundary Value Problem

$$\Rightarrow u_{i,j} = \frac{\Delta y}{\Delta x + \Delta y} u_{i-1,j} + \frac{\Delta x}{\Delta x + \Delta y} u_{i,j-1}$$

$$\Rightarrow u_{i,j} = C_x u_{i-1,j} + C_y u_{i,j-1}$$

Wavefront Parallelization: The domain

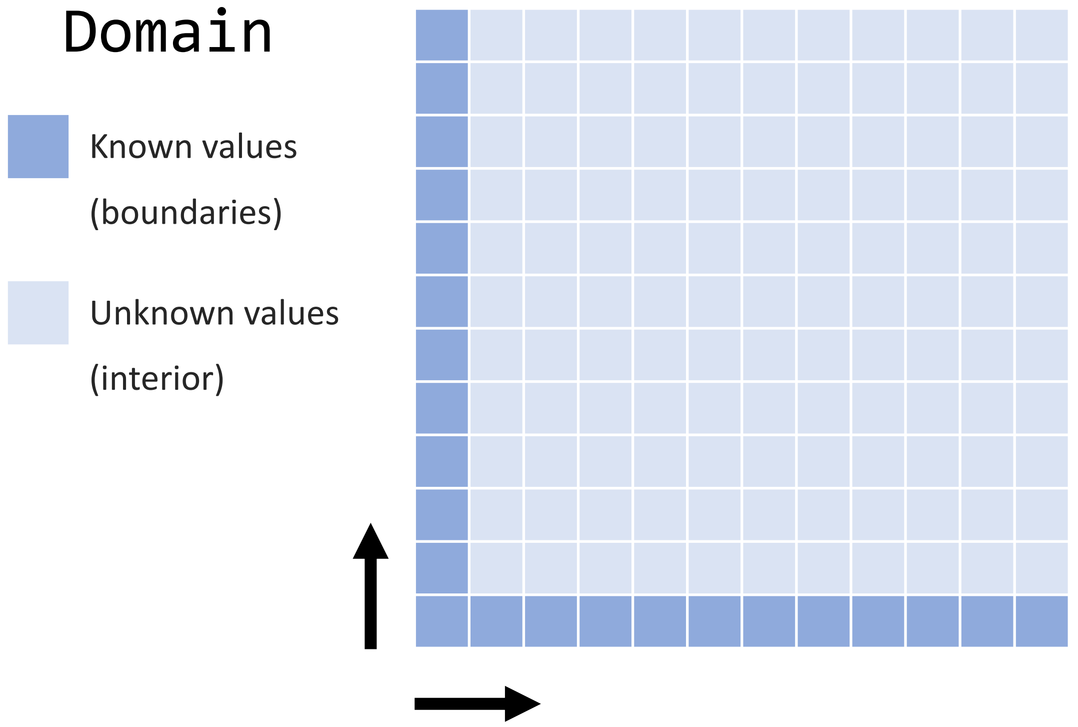
Example: 2D Boundary Value Problem

- The PDE gives us the equation to be applied, i.e., the actions to be done
- The threads are the workers
- How much work does each worker get?
  - How much work is there to be done?

    Work = the size of the domain = nx * ny
  - Load balancing: we'd like an even amount of work/worker
  - There are multiple ways to ***decompose the domain*** with constant work per worker, but some are better than others

The BVP: What We Need to Do

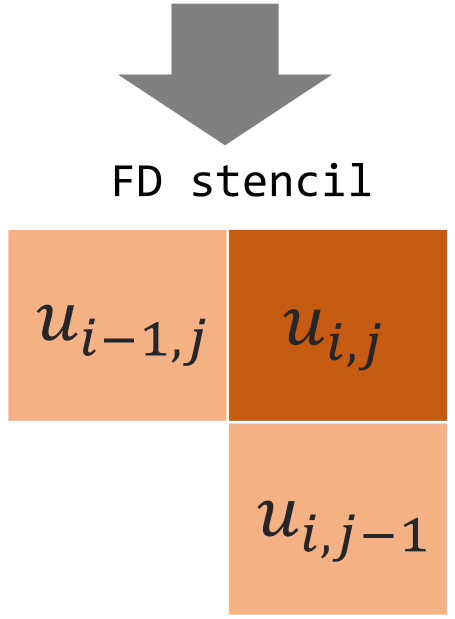
- We want to spread information on the boundaries inward



$$u_{i,j} = C_x u_{i-1,j} + C_y u_{i,j-1}$$

We have to fill in nodes from the lower left corner outwards due to the stencil's dependencies

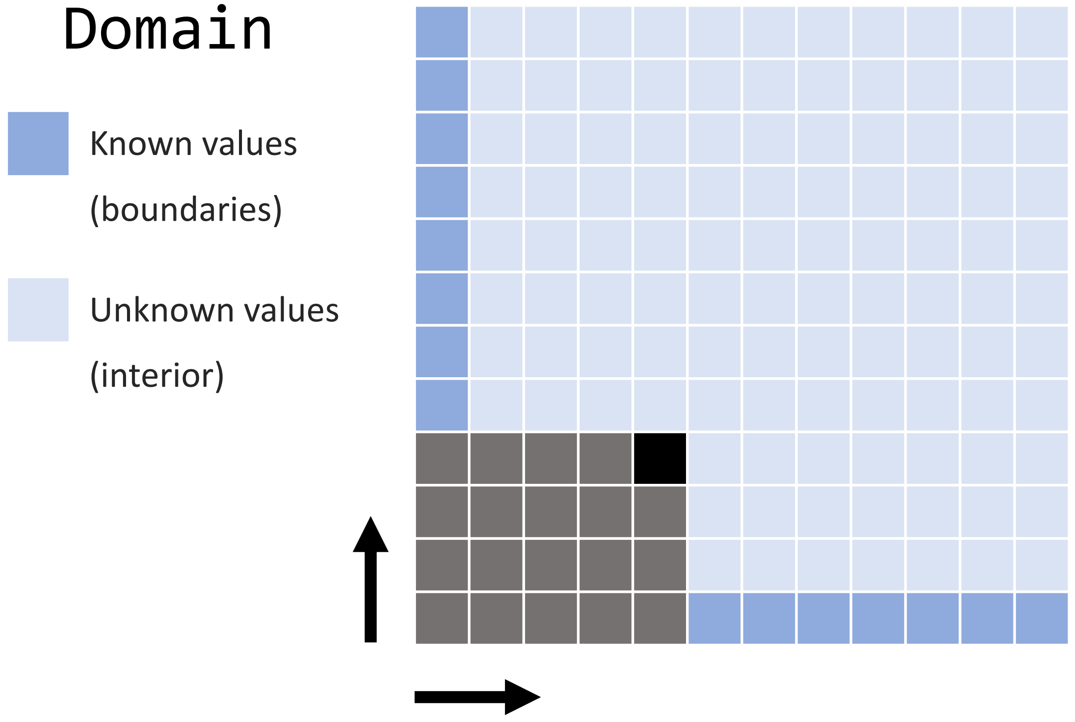
Note: we still have to fill in the boundary nodes too (for our implementation), they just don't depend on other nodes



The BVP: The Stencil’s Order Depenency

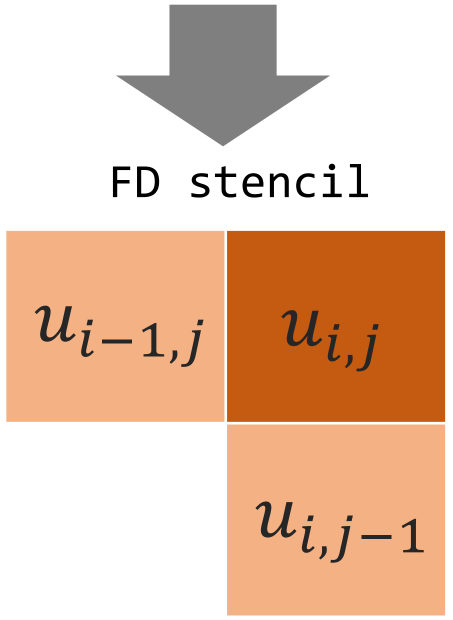
- We want to spread information on the boundaries inward

Domain



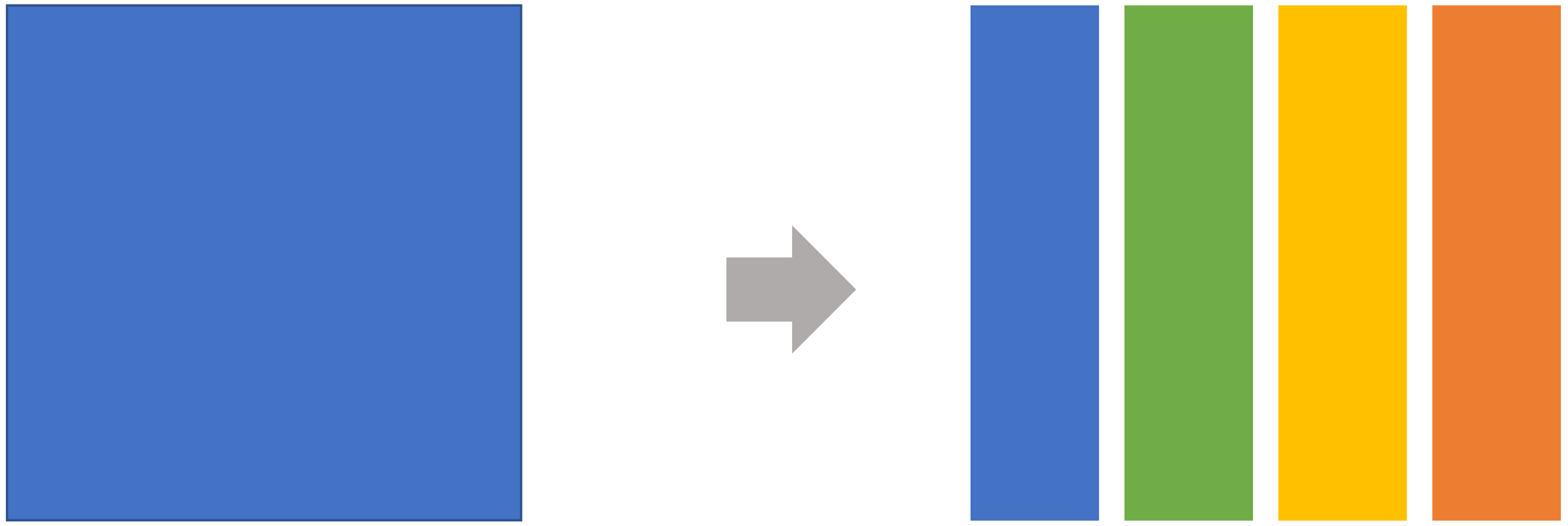
$$u_{i,j} = C_x u_{i-1,j} + C_y u_{i,j-1}$$

Example: can’t initialize the black node until all of the grey nodes have been initialized



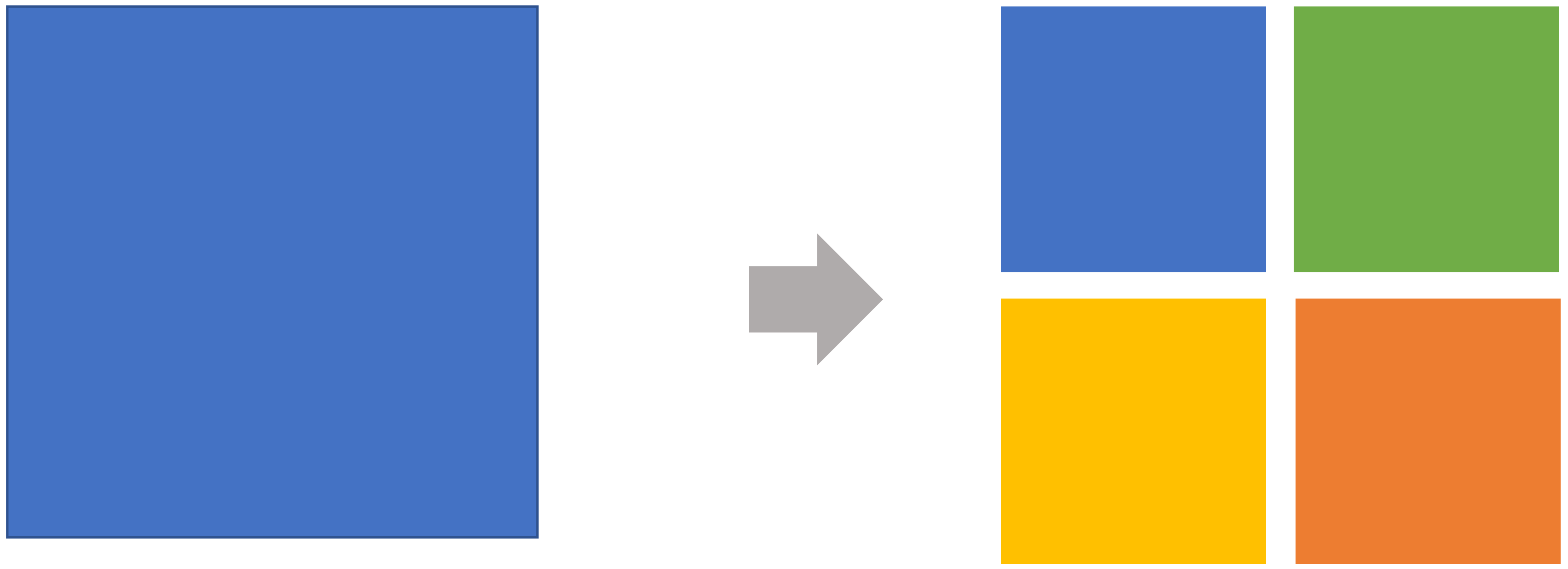
Domain Decomposition 1

Domain Decomposition 1: Slices



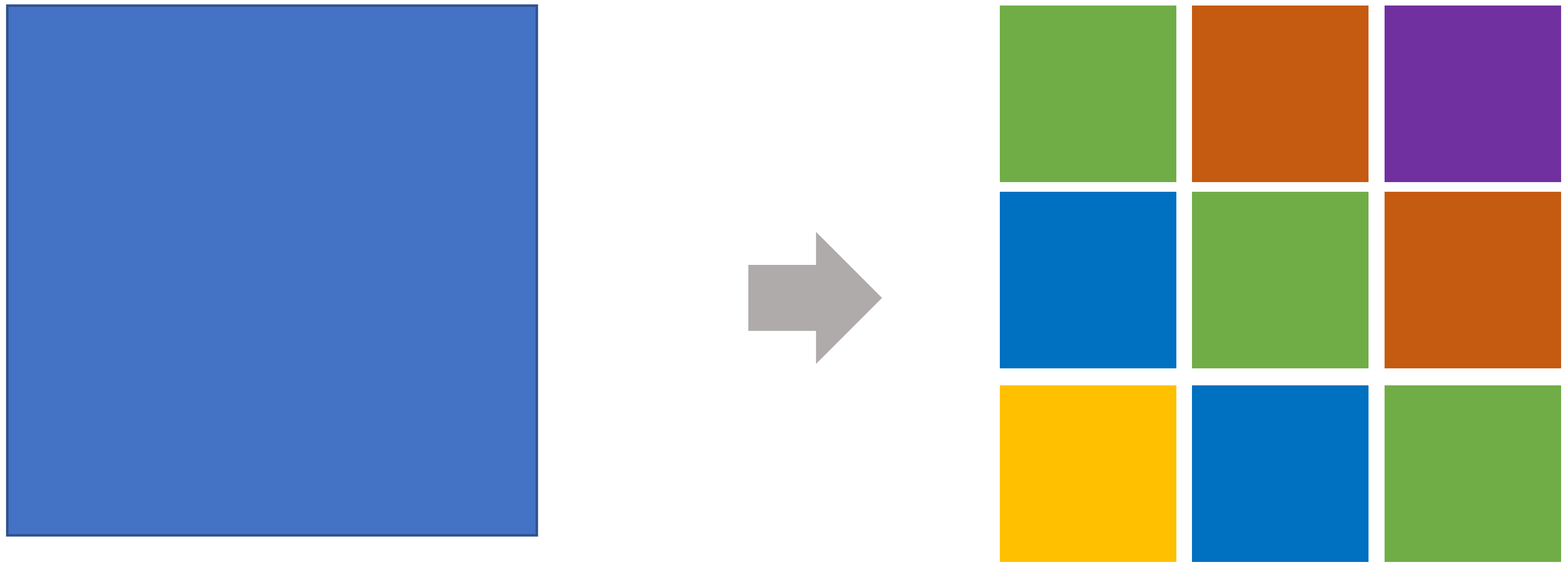
Domain Decomposition 2

Domain Decomposition 2: Cubes



Domain Decomposition 2

Domain Decomposition 2: Cubes



Domain Decomposition

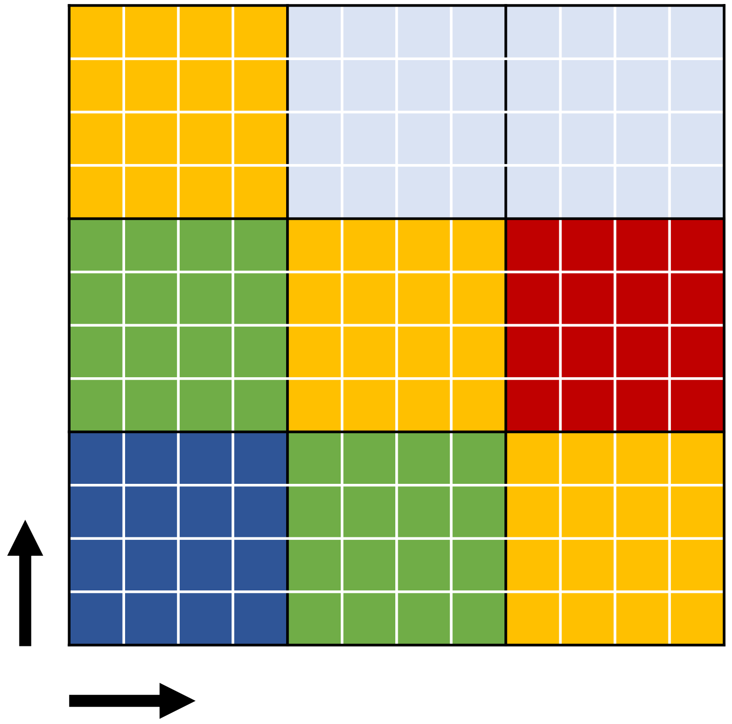
- **The slices CAN’T be parallelized!**
  - The order dependency of the stencil prevents it

- **The cubes CAN be parallelized**
  - Having multiple dimensions and defined boundaries let’s us process SOME cubes at the same time

- Both had constant work/worker, but one actually allows parallelization

The BVP: The Parallelization

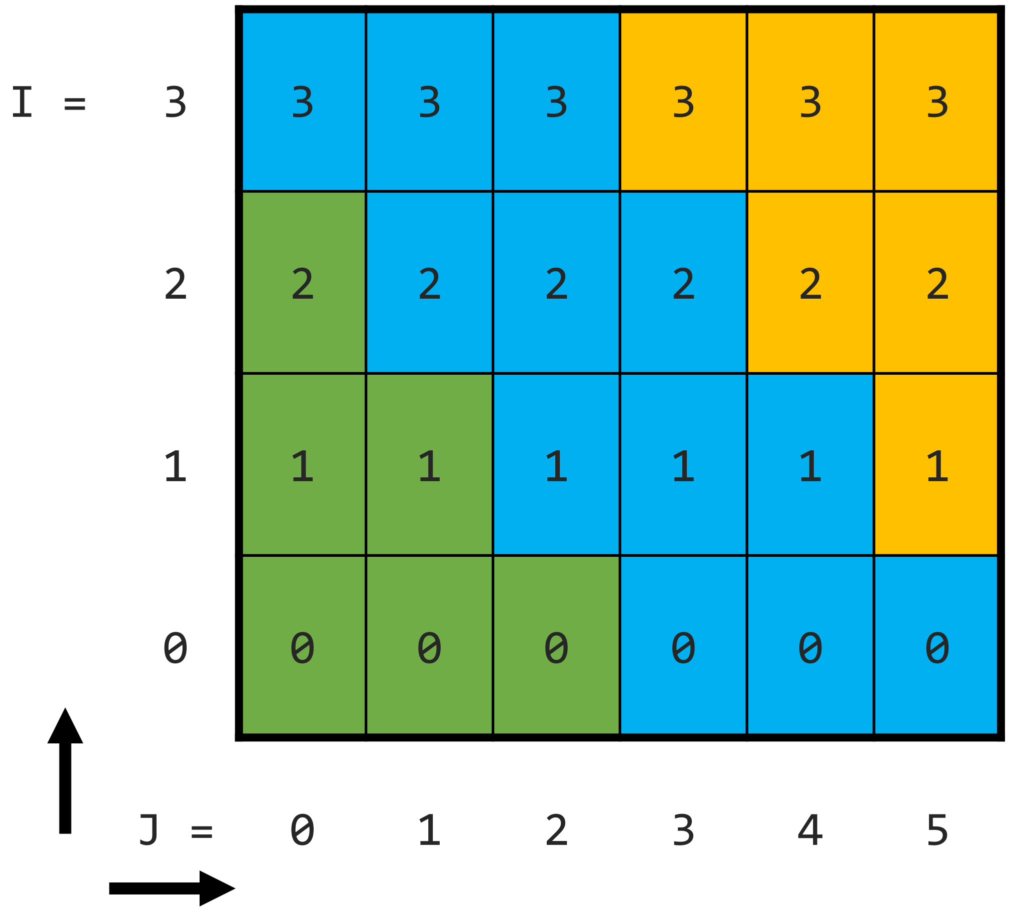
- Blocks on the same diagonal can be done in parallel



Example: Once the dark blue block is done, the two green blocks can be done in parallel, then the gold, etc

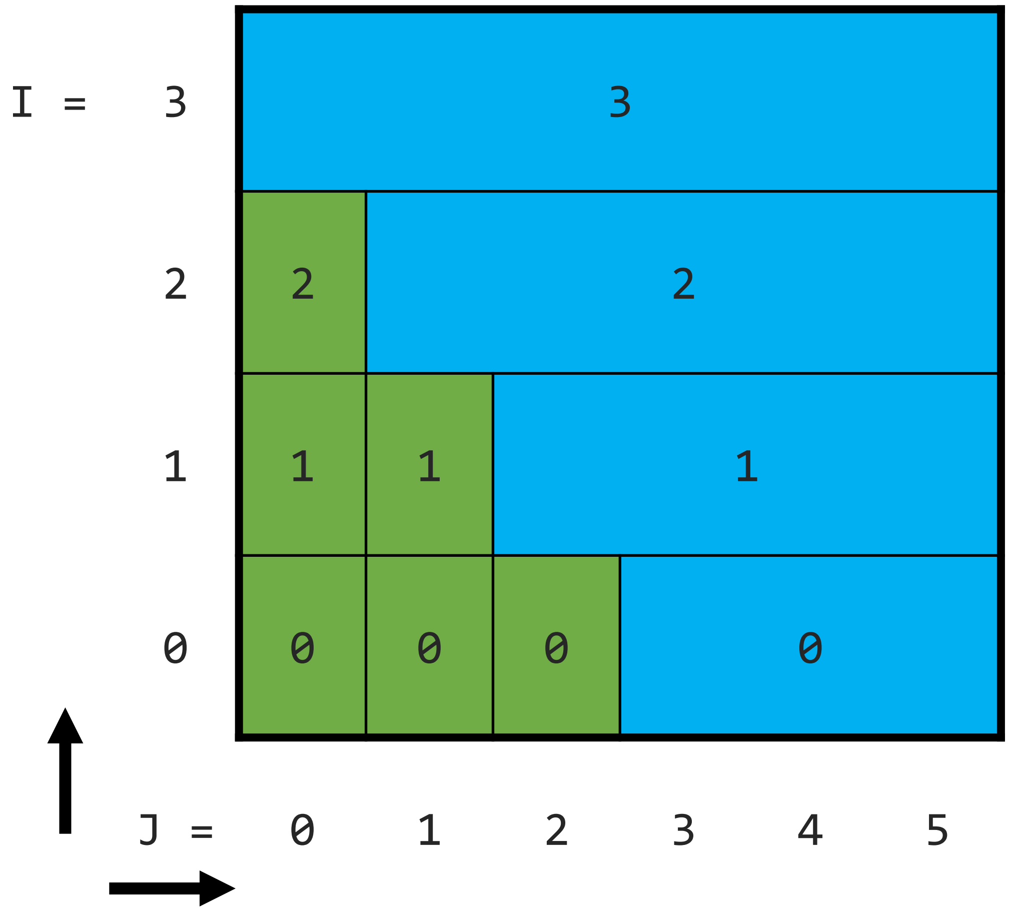
But notice: we could process the red block and the **uppermost** gold block at the same time, but the red block **must be done after** the lower two gold blocks

Simple Version: Choose NI or NJ to = NT



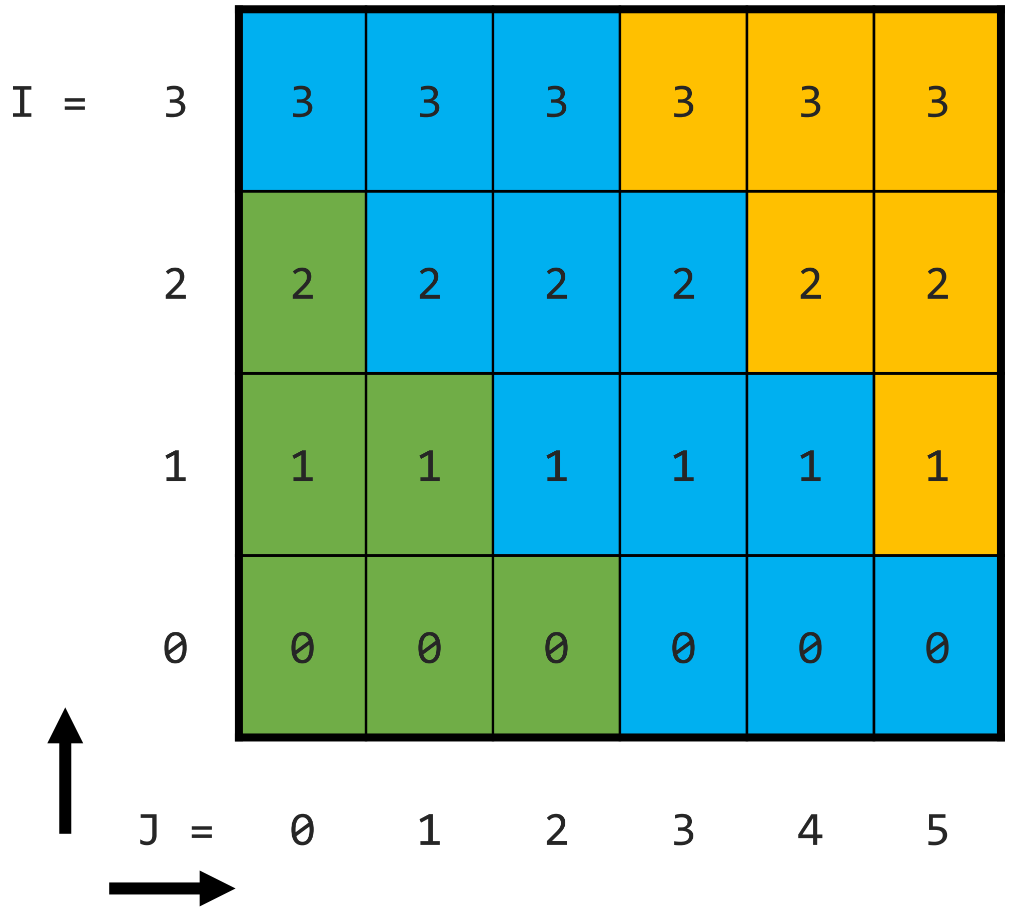
- **Pros:**
  - **Simple => easy to implement, debug, and maintain;** blocks can be any size so long as they're uniform
- **Cons:**
  - **The spin-up and spin-down phases take up more of the domain**
    - Smaller blocks help us reach the fully spun-up phase faster + get better parallelism
    - Still have some control: if NI = NT, pick NJ to have smaller blocks

Simple Version: Invalid Doppleganger!



- **This is not equivalent to the previous domain decomposition!**
- There is no way to synchronize in the J/j direction
  - Threads can start at different times and get out of step with one another
- The order dependency is in both I/i and J/j
- The diagonals act as synchronization points

Simple Version: Implementation Notes



- Observe that:
  - We need to synchronize and iterate along the diagonals
  - On a diagonal d, I + J = d
  - There are NI + NJ – 1 diagonals
  - I corresponds to the thread ID (for NI = NT)
- **Can iterate on d rather than I and J**

Simple Version: Code Sketch

- When exploiting the problem’s structure, the code is simple

```
#pragma omp parallel // Not a parallel-for!
{
    int I = omp_get_thread_num();
    int J;
    for(int d = 0; d < NI + NJ - 1; d++){
        J = d – I;
        if( J > 0 && J < NJ ) // Make sure we’re in bounds
            process_block(…);
        #pragma omp barrier
    }
}
```