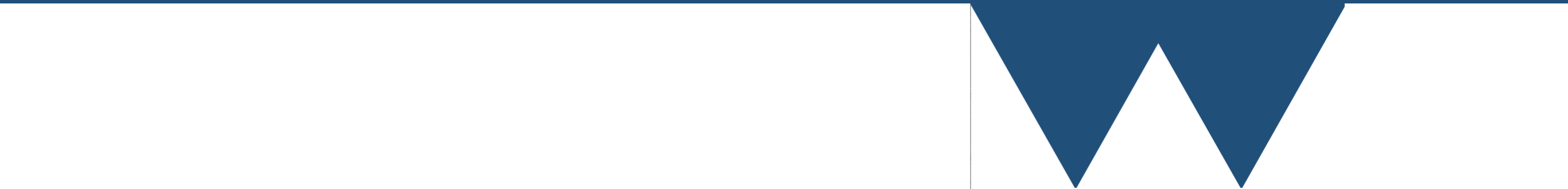


Threads and OpenMP

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.



Is Threading for Me?



Do you want Threading?

- Threads and OpenMP lets you make use all of the processors on a single machine
- They **don't** let you run your code across multiple computers
 - They don't give you any extra memory to do your work in
 - You can write programs that run across multiple computers and hence give you extra memory but this is NOT that

Concepts



Concepts

- When you write a normal computer program you are giving a sequence of instructions that a processor works through one one after the other
- If you want to use multiple processors you either have to
 - Perform different *instructions* on different *data*
 - Perform different *instructions* on the same data
 - Perform the same instructions on different *data*

Concepts

- When you write a normal computer program you are giving a sequence of instructions that a processor works through one one after the other
- If you want to use multiple processors you either have to
 - Perform different *instructions* on different *data*
 - Pe Loop parallelism - easy mode data
 - Perform the same instructions on different *data*

Concepts

- When you write a normal computer program you are giving a sequence of instructions that a processor works through one one after the other
- If you want to use multiple processors you either have to
 - Task parallelism - easy mode - any paradigm
 - Perform different *instructions* on different *data*
 - Perform different *instructions* on the same data
 - Perform the same instructions on different *data*

Concepts

- When you write a normal computer program you are giving a sequence of instructions that a processor works through one one after the other
- If you want to use multiple processors you either have to
 - Perform **Task parallelism - hard mode** *ta*
 - Perform different *instructions* on the same data
 - Perform the same instructions on different *data*

Note on Python



Note on Python

- Python is a very popular language but it is **not** well suited to parallel programming
- Python's native parallelism model is a threading model
 - Until recently the normal Python interpreter didn't actually doesn't work in parallel! (Others like PyPy didn't have this problem)
 - Called the Global Interpreter Lock (GIL)
 - The 3.14 release in October 2025 removed the GIL
 - Multithreaded performance in Python is still not good

Note on Python

- Libraries that Python code uses can exploit threading with much higher efficiency
- **Write as little Python code as you can!**
- Read the docs on your library to see how to tell it how many threads to run
- Not much else that is under your control

Concepts



Concepts

- Threading approaches exploit the fact that memory is shared
- Both a strength and a weakness of this type of programming
- The problem is very much like multiple people working to clear a table
 - If you all take different plates then there's no problem
 - If two people try to take the same plate then there's a collision
 - In general you get the fastest result if none of these collisions occur
 - **Things are actually worse with computers because collisions can also give you the wrong answer!**

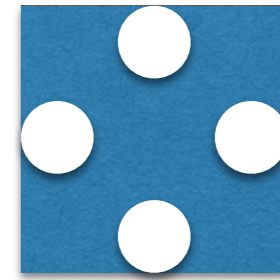
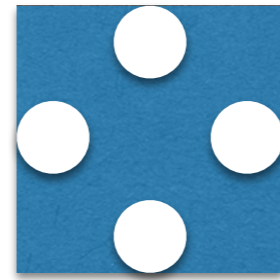
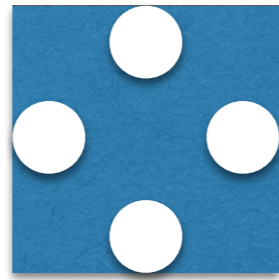
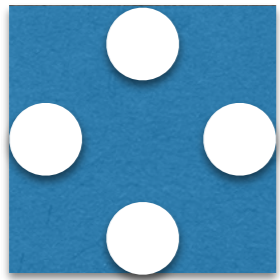
Concepts

- Formally this is described through **Bernstein's Conditions**
- You can parallelise two operations O1 and O2 so long as
 - O1 doesn't use any data that O2 modifies
 - O2 doesn't use any data that O1 modifies
 - O1 and O2 don't try to modify the same data
- You can mix operations where these conditions are satisfied with conditions where they aren't but you can't do operations that don't satisfy these conditions **in parallel**

Concepts

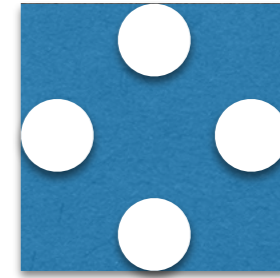
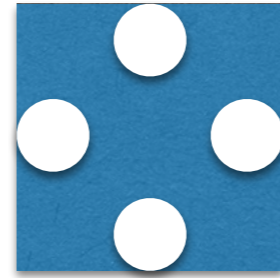
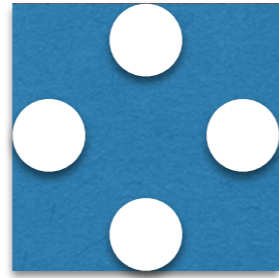
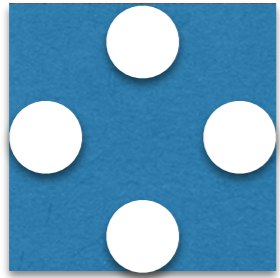
- Ensuring that this is true is a large part of the skill of parallel programming
- You want to make sure that you separate your job into problems that don't interfere with the memory that the other problems are trying to use
- This sounds hard but is often quite simple
 - Sometimes even automatic

Table Clearing



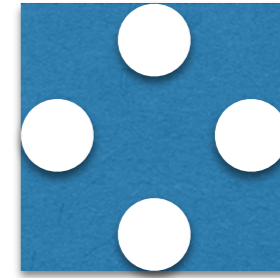
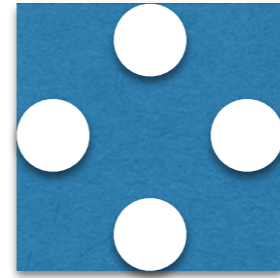
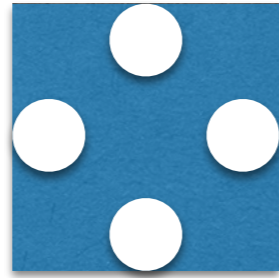
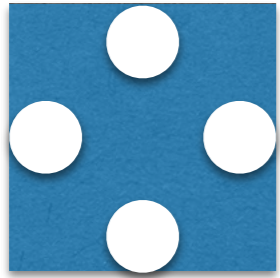
- Imagine the table clearing idea again
- Two clearers trying to clear a single place will cause a collision
- There are various strategies

Table Clearing



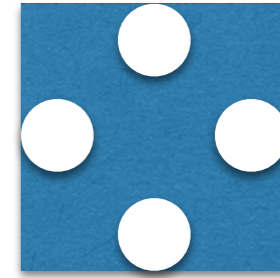
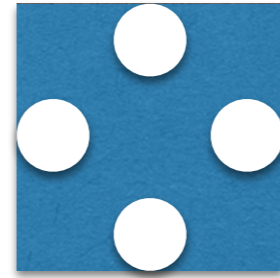
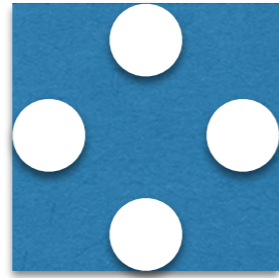
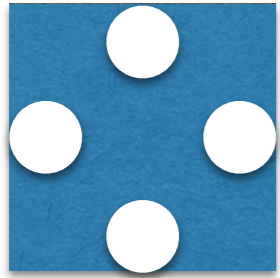
- Assign every clearer a single table
 - or half table
- One clearer per place

Table Clearing



- There are bad choices though
 - Customer wearing blue vs not wearing blue is badly load balanced (two servers but one will probably have more work than the other)

Table Clearing



- Wearing blue vs wearing earrings has possible collisions
- Can collide if someone is wearing blue and wearing earrings

Table Clearing

- Most of the strategies that seem sensible will be fairly sensible
 - They may not be optimal
- This also applies to a lot of problems in actual parallel programming
 - Especially if you're retro-fitting to existing code!

Thread Safety

- The key issue is "thread safety"
 - Can a function or operation be safely done by more than one thread at once?
 - Will it get the right (same as before*) answer if it is?
 - Does it modify global or shared state?
 - Does it use library code which does (might)?
 - Does it store information which might become invalid (e.g. iterator invalidation)

Computational Problem

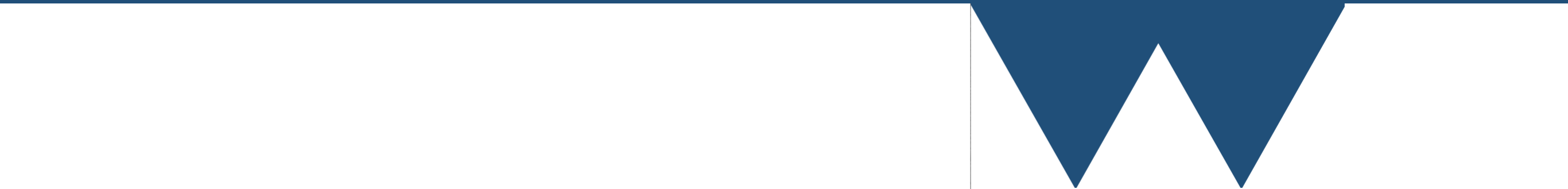
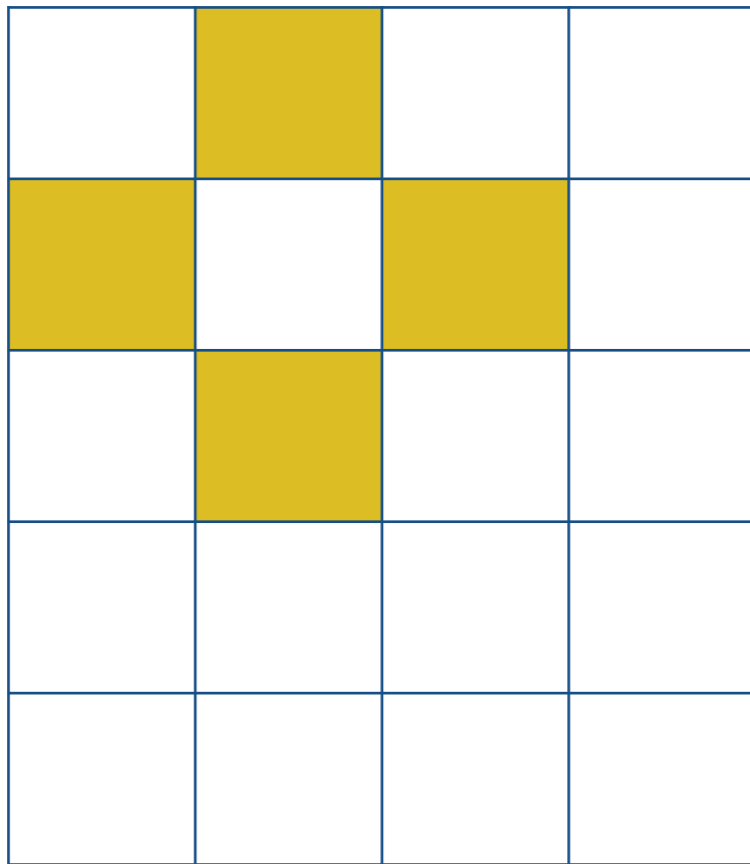


Image Blurring

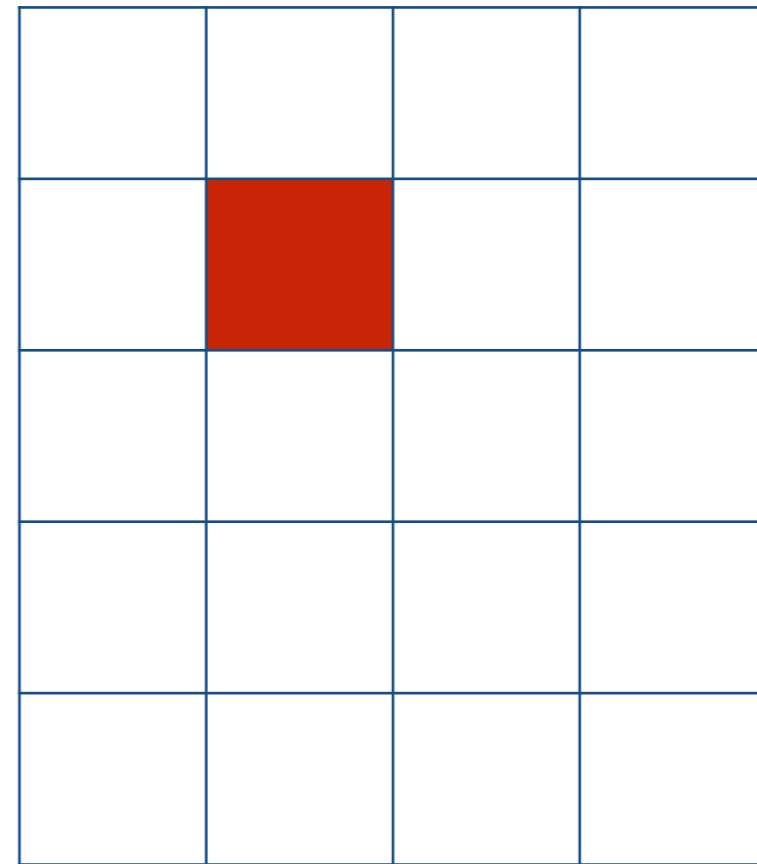
- There are lots of ways of blurring images
- The simplest way is to replace the value in each pixel with the average of the values in the four surrounding cells
- You don't update a pixel with the average value immediately but store that average value to a temporary array and then copy the temporary values back to the real array in at the end

Image Blurring

Main Array



Temporary Array



- Yellow cell = pixels in main array used to update the red cell in the temporary array
- Red cell = new value in the temporary array

Image Blurring

- Any 'threading' based approach will be pretty similar
- As long as blurred image is a new copy everything is easy
- All threads READ from the same original image
- Each thread writes to its own output pixels
- Bernstein's conditions satisfied

Image Blurring

- With plain threads, everything is up to us
- We decide how to split the work (rows, columns, blocks)
- Optimisation would mean having a thread read the same pixels as often as possible
- We ensure the final writes do not overlap
- We take care of saying when we're done

Image Blurring

- OpenMP is *designed* for this sort of thing
 - Largely because this is hugely common in the problems where it was first used
- If you're in C, C++ or Fortran, it's absolutely the way to go for this sort of loop splitting
- Tons of actual code is in the accompanying repo
- Example for this case is super easy

Image Blurring

```
!$OMP PARALLEL DO
  DO iy = 1, UB1
    DO ix = 1, UB2
      temp(ix,iy) = 0.25 * (image(ix+1,iy) + image(ix-1,iy) &
        + image(ix,iy+1) + image(ix,iy-1))
    END DO
  END DO
!$OMP END PARALLEL DO
image = temp
```

Image Blurring

```
!$OMP PARALLEL DO
```

```
DO iy = 1, UB1
```

```
DO ix = 1, UB2
```

```
temp(ix,iy) = 0.25 * (image(ix+1,iy) + image(ix-1,iy) &  
+ image(ix,iy+1) + image(ix,iy-1))
```

```
END DO
```

```
END DO
```

```
!$OMP END PARALLEL DO
```

```
image = temp
```

The OpenMP directives (comments in the source language)

Differ a bit in C and Fortran

Image Blurring

```
!$OMP PARALLEL DO
  DO iy = 1, UB1
    DO ix = 1, UB2
      temp(ix,iy) = 0.25 * (image(ix+1,iy) + image(ix-1,iy) &
        + image(ix,iy+1) + image(ix,iy-1))
    END DO
  END DO
!$OMP END PARALLEL DO
image = temp
```

IMPORTANT: the copy back is OUTSIDE the OMP bit

Image Blurring

```
!$OMP PARALLEL DO
  DO iy = 1, UB1
    DO ix = 1, UB2
      temp(ix,iy) = 0.25 * (image(ix+1,iy) + image(ix-1,iy) &
        + image(ix,iy+1) + image(ix,iy-1))
    END DO
  END DO
!$OMP END PARALLEL DO
image = temp
```

Something to think about: how much of the work is parallel here? Could we do more?

Easy OpenMP

- As well as loops, the following things are pretty easy to do:
 - Serialise threads where needed (ATOMIC, CRITICAL)
 - Control shared and non shared data (PUBLIC and PRIVATE)
 - Parallelise operations and perform a final reduce (REDUCTION)
 - Parallelise Fortran whole array operations (WORKSHARE)