Data Parallelism

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Task Parallelism

- In our image processing application (release 1.4), each processing thread works on a different image
- This was the only thing we could do really, since the BufferedImageOp.filter() method is not multi-threaded
- What we did is called task parallelism
  - Each thread does a different thing, or the same thing but on different data objects (in our case, an image)
- But what if a user wants to use the app to process a single large image???
  - Which could take a long time (e.g., oil filter)
- Our app uses a single core for the computation, meaning the user’s other cores are “wasted”

Data Parallelism

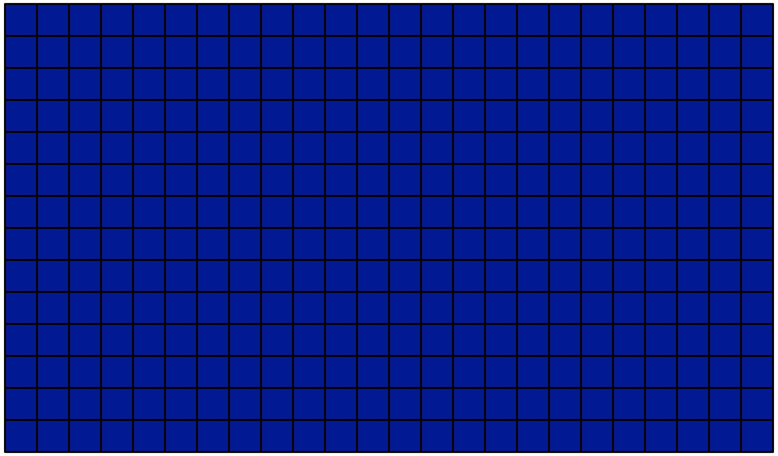
- We need to have multiple threads work on the same image!
- This is typically called data parallelism
  - Threads apply the same exact computation to different elements within a dataset
  - e.g., different threads apply the same computation to the pixels of the same image
- The distinction between task and data parallelism is a bit blurry
  - If we consider a set of images to be our datasets, then having each thread working on an image can be construed as data parallelism
  - But typically, one uses the term data parallelism when each thread applies computation to small/scalar items (pixels, array elements, etc.)
- Both kinds of parallelism can occur within the same application

Stencil Applications

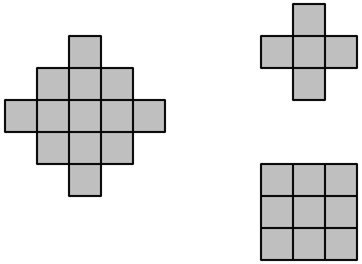
- Many useful applications can benefit from data parallelism
- A classical example is stencil applications
- An application operates on some “domain” (basically an array), and updates each element based on the value of neighboring elements
  - Perhaps multiple times in sequence
- Let’s see this on a picture for a 2-D domain…

Stencil Application Basics

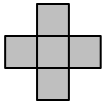
2-D domain



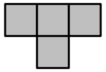
Example stencil shapes



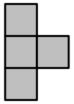
Blurring effect



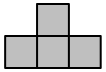
d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) / 5



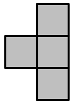
d[i][j] = (d[i][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) / 4



d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j+1]) / 4

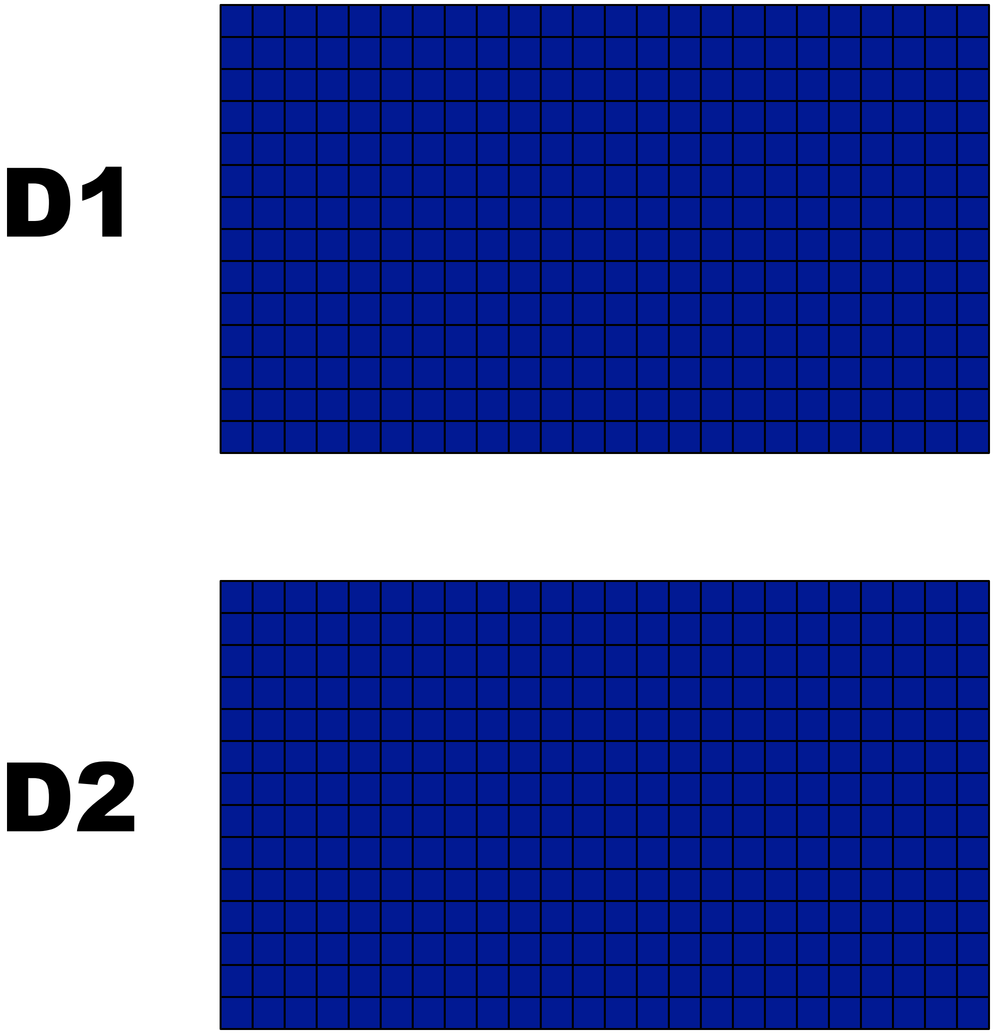


d[i][j] = (d[i][j] + d[i-1][j] + d[i][j-1] + d[i][j+1]) / 4



d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j-1]) / 4

Stencil Application Basics

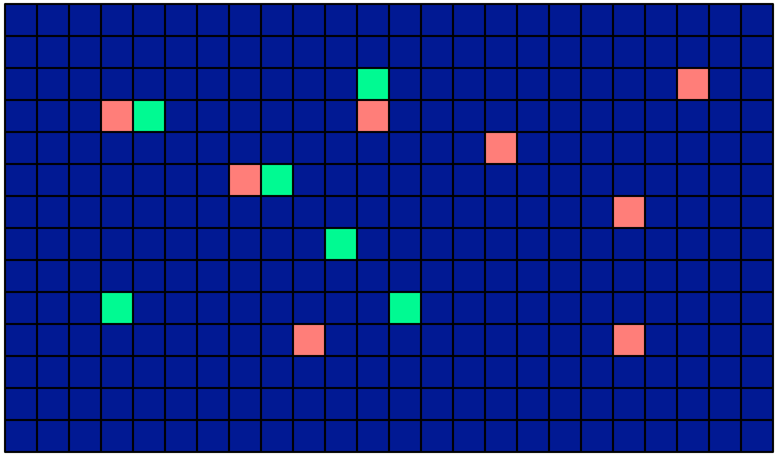


Create two domains

(i.e., two arrays in memory)

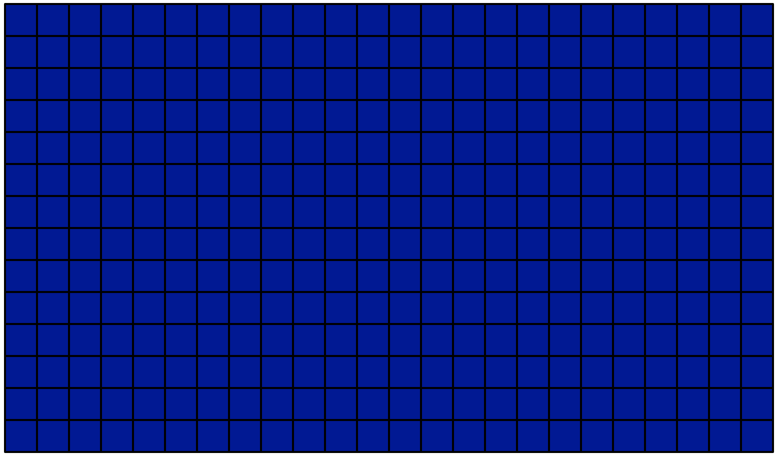
Stencil Application Basics

D1

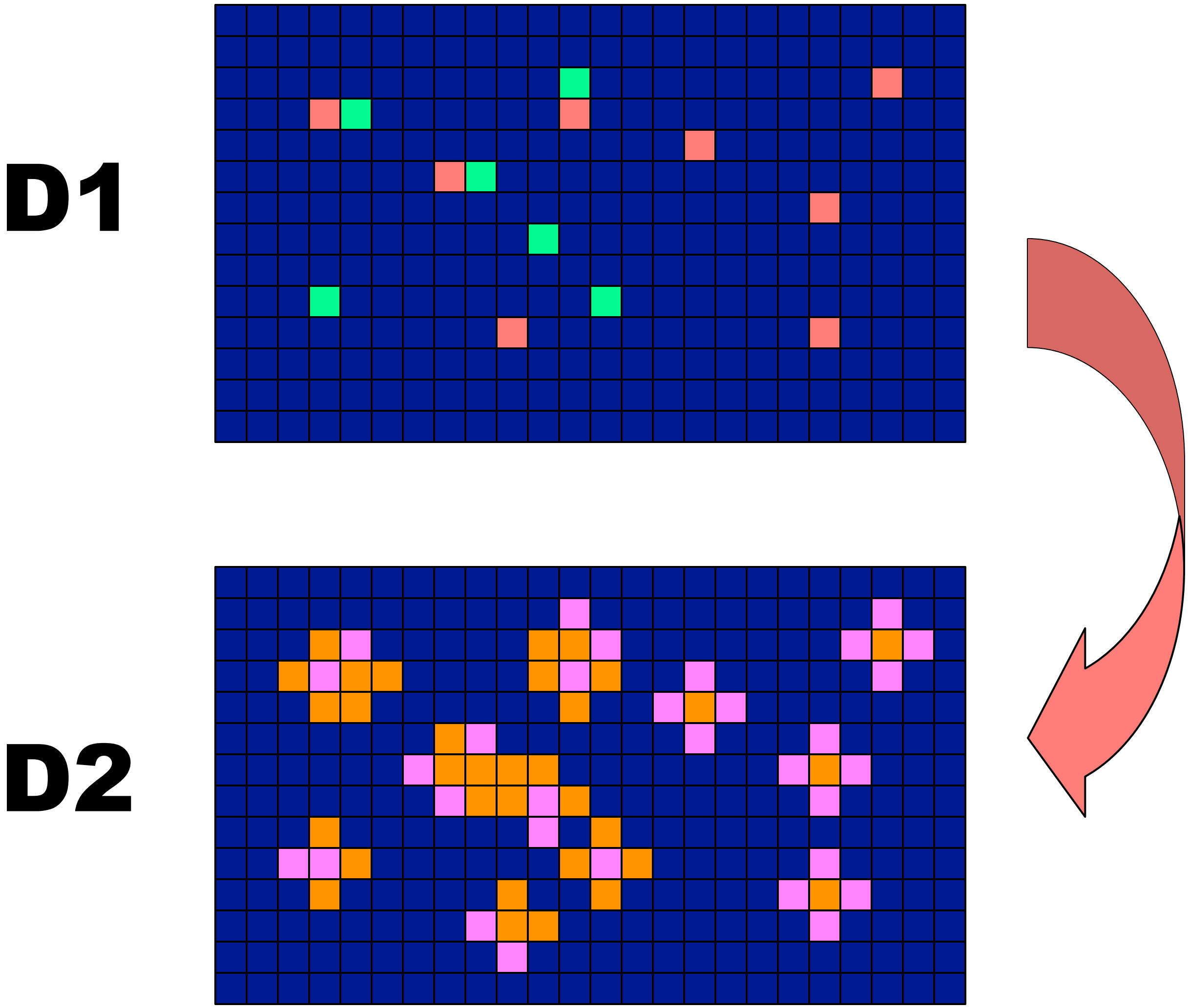


Store some initial values in D1

D2



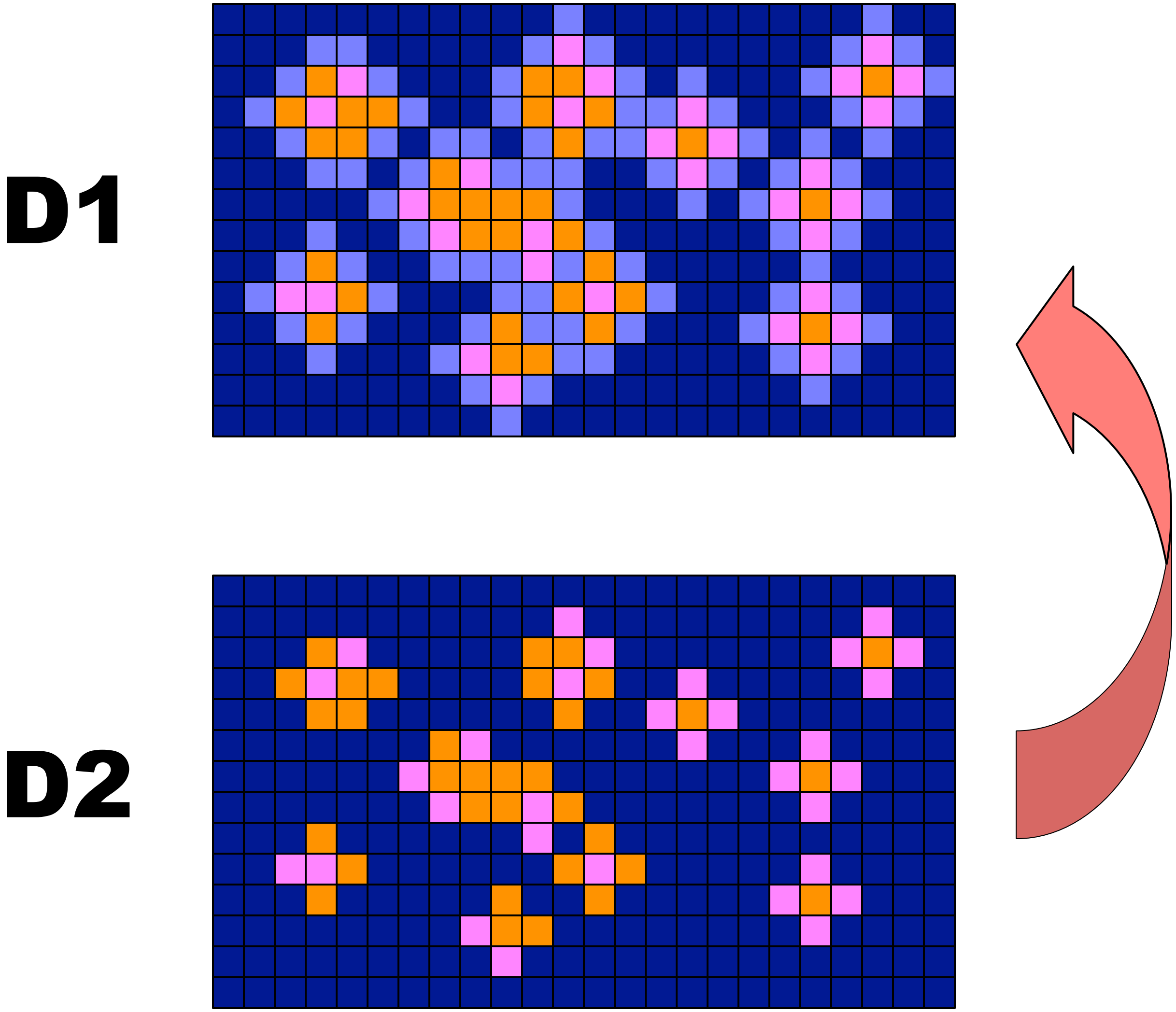
Stencil Application Basics



Compute values in D2 based on values in D1

D2 holds values at iteration 1

Stencil Application Basics



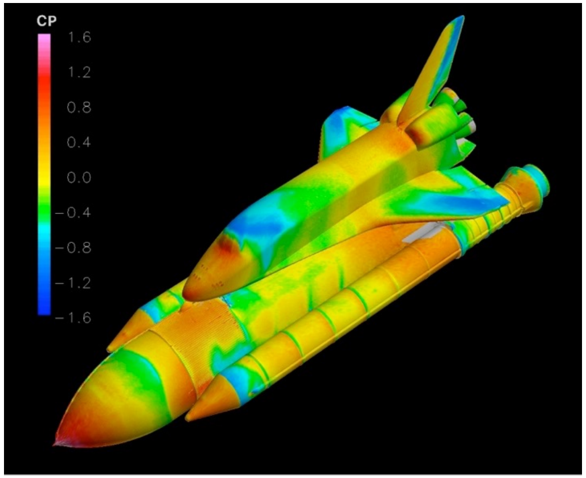
Compute values in D1 based on values in D2

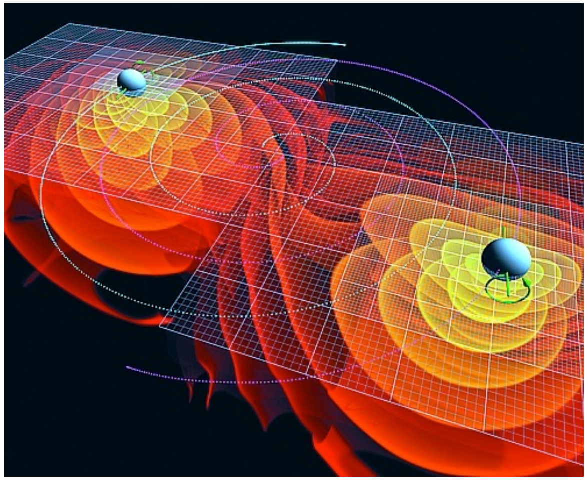
D1 holds values at iteration 2

and so on...

Stencil Applications Galore

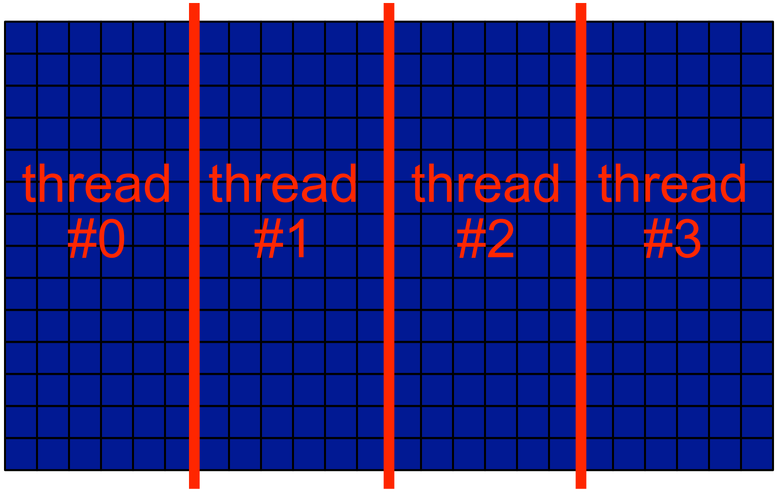
- Many useful computations are stencil applications
- Computational fluid dynamics, convolution filters for image processing, physics, deep learning, etc.





Multi-threaded Stencil Apps

- Because all element updates are independent of each other, a stencil application is easy to parallelize using multiple threads
- Split the domain into "slabs" and have each thread compute elements in on of these slabs

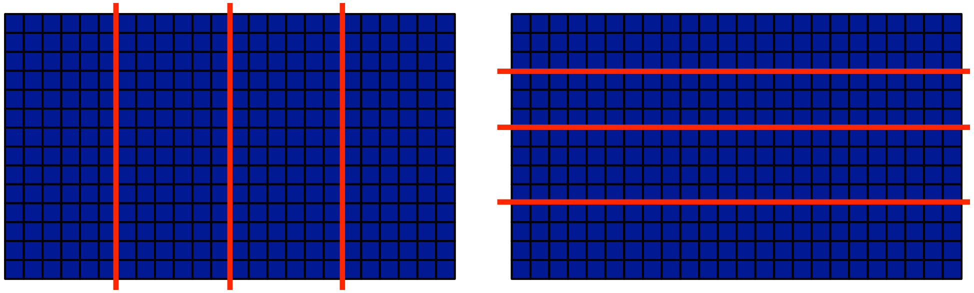


- 4 cores
- 24x14 domain
- each thread processes a 6x14 slab

- Synchronize all threads (barrier) before moving on to the next iteration

Domain Decomposition

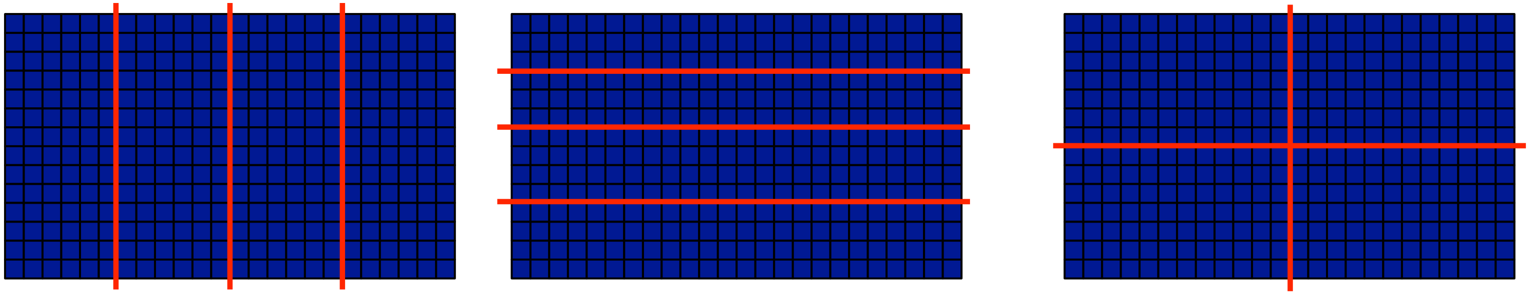
- There are many options for “domain decomposition”, i.e., dividing the work among threads



14 is not divisible by 4, so load balance is not perfect!

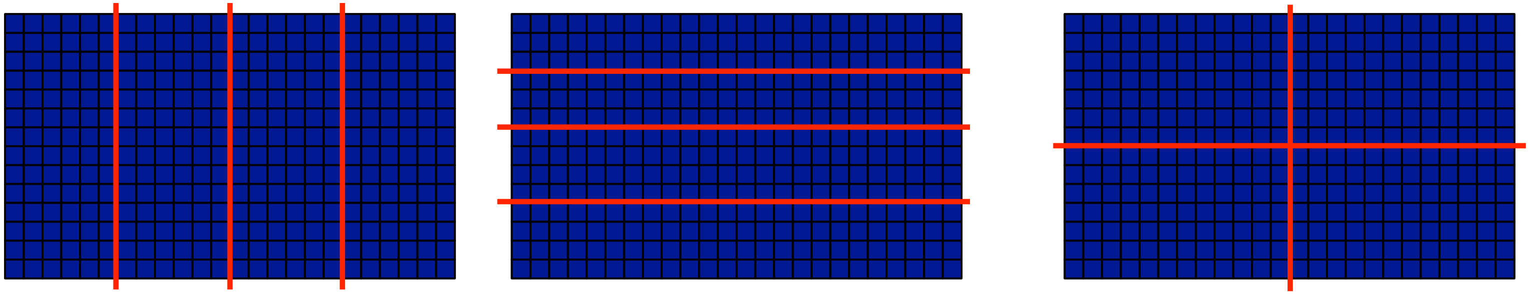
Domain Decomposition

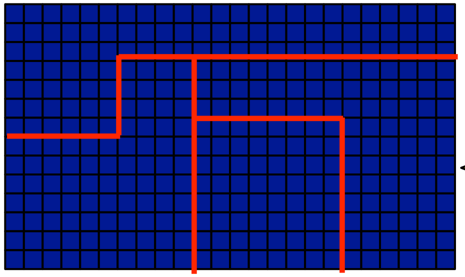
- There are many options for “domain decomposition”, i.e., dividing the work among threads



Domain Decomposition

- There are many options for “domain decomposition”, i.e., dividing the work among threads

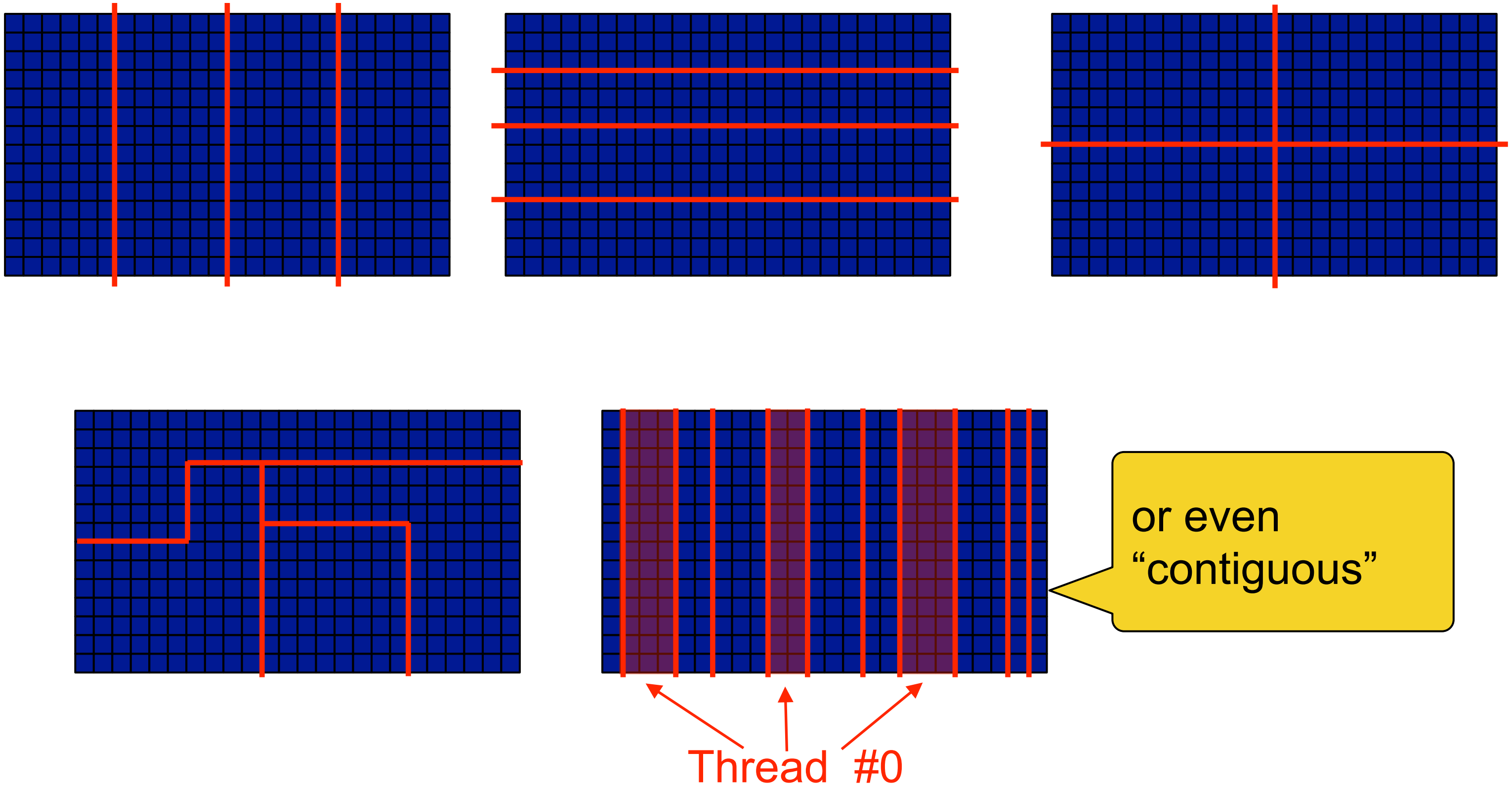




Doesn't have to be “regular”

Domain Decomposition

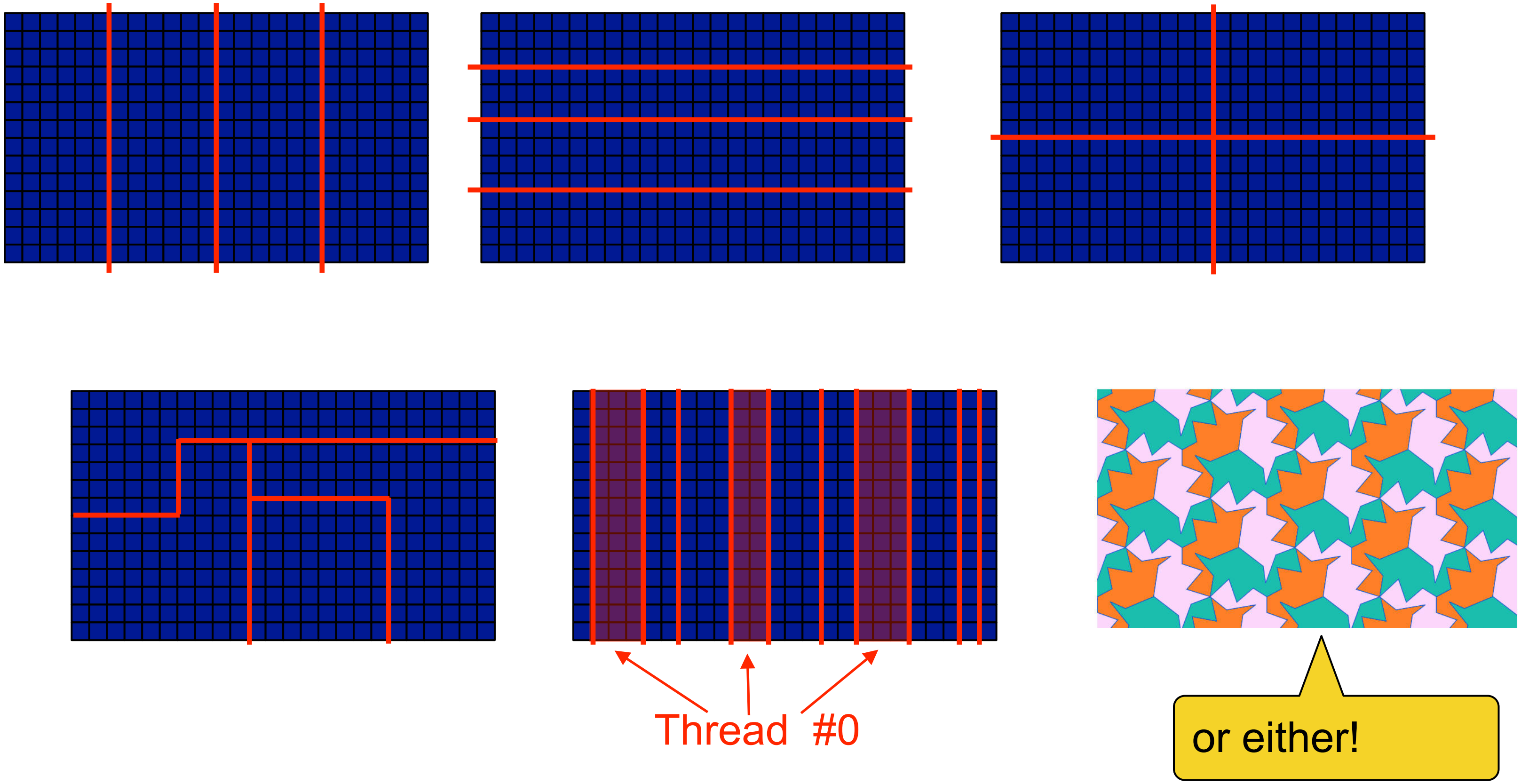
- There are many options for “domain decomposition”, i.e., dividing the work among threads



or even “contiguous”

Domain Decomposition

- There are many options for “domain decomposition”, i.e., dividing the work among threads



Domain Decomposition

- There are advantages / drawbacks to different domain decomposition schemes
- Some of them may boost performance
  - Due to “locality” (stay tuned)
- Some of them are more difficult to implement than others
  - You have to write code to figure out “If I am thread #i, am I in charge of element (x,y)?”
    - Could be trivial discrete math (e.g., horizontal slabs)
    - Could be very complicated (e.g., gerrymandering)

SPMD: Single Program Multiple Data

- Threads have an ID and based on their IDs they should know what to compute
  - This is all implemented by the programmer
- For instance, if we have two threads work on an array of N elements, we could write the thread code as:

```
for (int i=0; i < N; i++) {
  if  (i % 2 == my_id) {
    // Do the work for iteration i
  }
}
```

- This is called Single Program Multiple Data (SPMD): all threads run the same program but they take different execution paths in it based on their IDs

Simple Thread Synchronization

- There is no need for critical section
  - Because all elements can be computed independently, and no two threads ever update the same memory location
  - All threads can just work on their piece of the domain without any lock, and wait for each other before proceeding to the next iteration (if any)
- This is good news for performance, since critical sections are parallelism killers, and thus performance killers

Concurrent vs. Parallel Programs

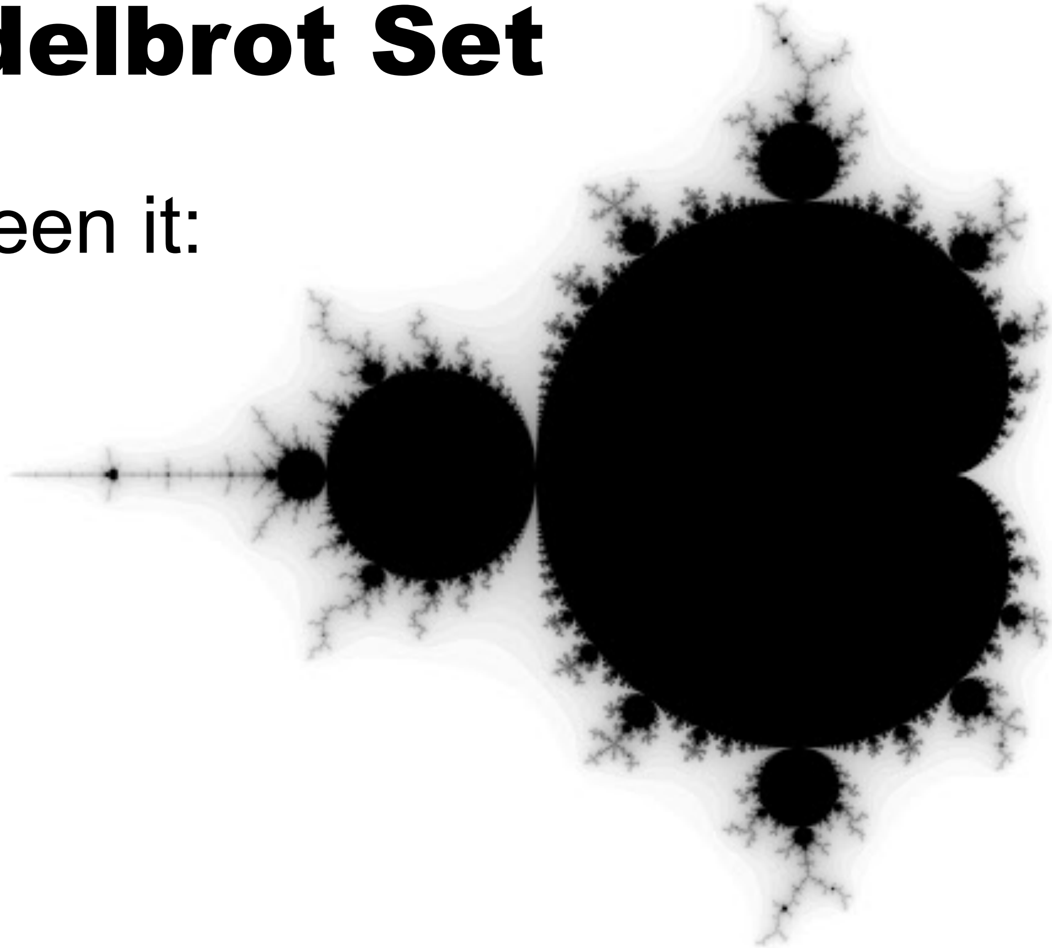
- Typically one draws the distinction between concurrent and parallel programs
- Concurrent program: We don't know what each thread will do ahead of time, but we know it will be correct because we implemented appropriate critical sections
- Parallel program: We know what each thread will do ahead of time, so we may be able to avoid using critical sections completely, which is better for performance
- We could implement a stencil application using concurrent computing
  - e.g., using producer-consumer by which threads answer the "what element should I process next?" question by grabbing the element (i.e., it's coordinates) from a producer-consumer buffer
- But it is not a good idea performance-wise if we can avoid it

Load Balancing

- In all the previous example, we have assumed that all element computations are identical
  - Each element of the domain is processed using the same number of arithmetic operations
- This is often the case, but not always
- Let’s look at a textbook example in which it is not the case…

The Mandelbrot Set

- You’ve all seen it:



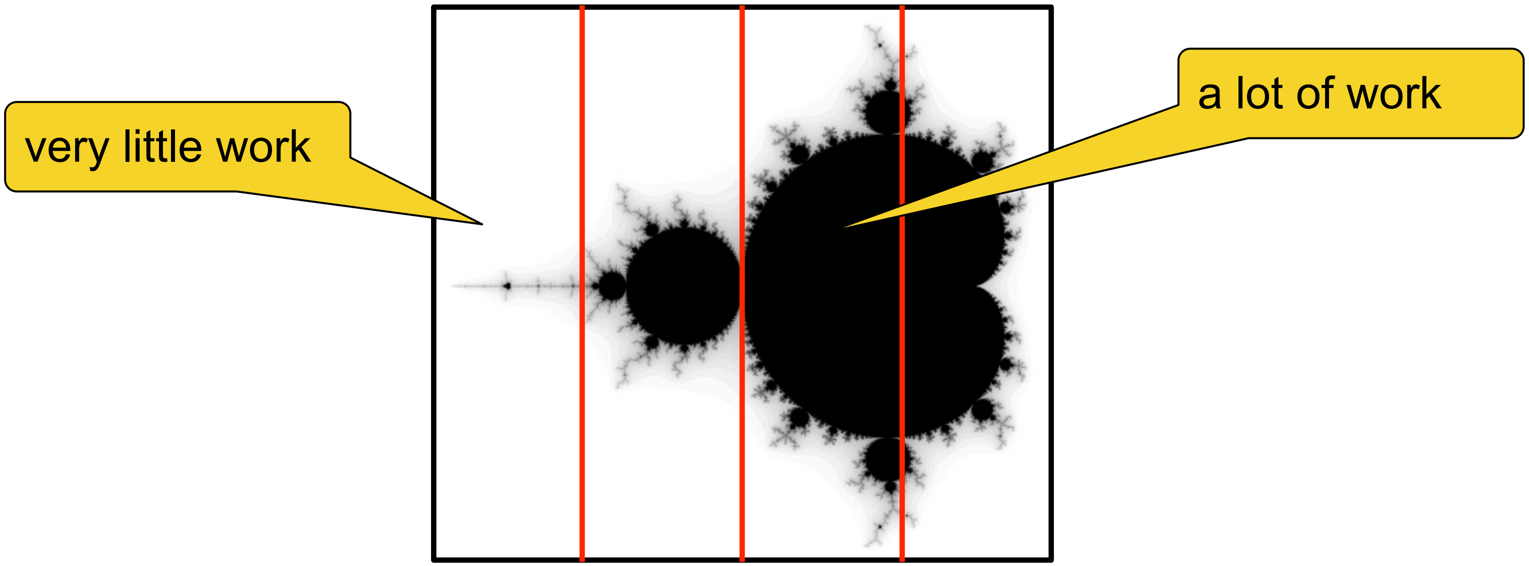
- It’s a textbook example of a stencil application in which not all elements are equal. Let’s see why….

Mandelbrot Set Definition

- For each complex number c, define the series
  - $Z_0 = 0$
  - $Z_{n+1} = Z_n^2 + c$
- If the series converges, paint the pixel at point c black
- If the series diverges, paint the pixel at point c white
- Determining convergence is typically more expensive than determining divergence (for Mandelbrot)
- So a thread that has more black pixels to process has more work to do!

Mandelbrot Set Definition

- For each complex number c, define the series
  - $Z_0 = 0$
  - $Z_{n+1} = Z_n^2 + c$
- If the series converges, paint the pixel at point c black
- If the series diverges, paint the pixel at point c white
- Determining convergence is typically more expensive than determining divergence (for Mandelbrot)
- So a thread that has more black pixels to process has more work to do!

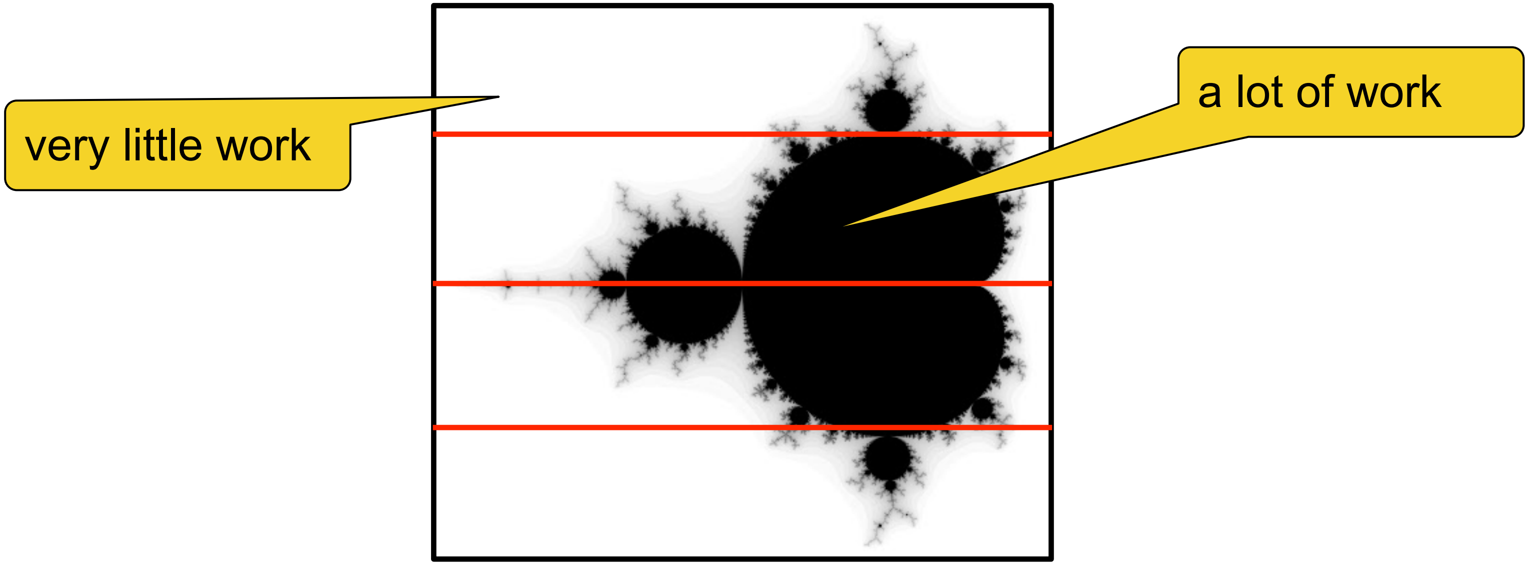


very little work

a lot of work

Mandelbrot Set Definition

- For each complex number c, define the series
  - $Z_0 = 0$
  - $Z_{n+1} = Z_n^2 + c$
- If the series converges, paint the pixel at point c black
- If the series diverges, paint the pixel at point c white
- Determining convergence is typically more expensive than determining divergence (for Mandelbrot)
- So a thread that has more black pixels to process has more work to do!

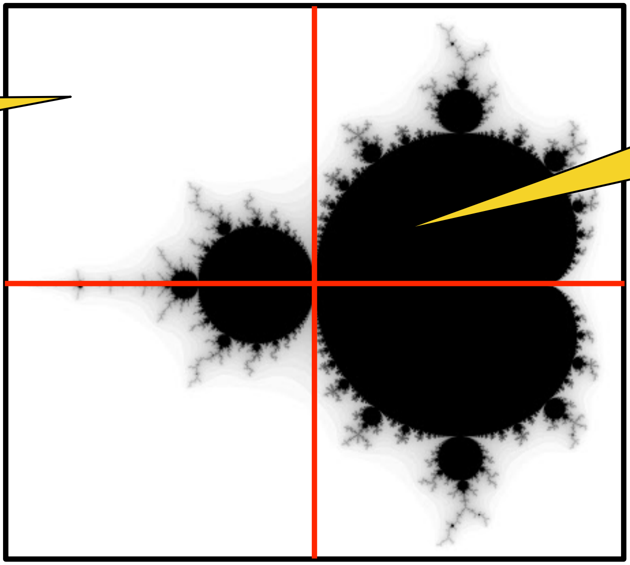


very little work

a lot of work

Mandelbrot Set Definition

- For each complex number c, define the series
  - $Z_0 = 0$
  - $Z_{n+1} = Z_n^2 + c$
- If the series converges, paint the pixel at point c black
- If the series diverges, paint the pixel at point c white
- Determining convergence is typically more expensive than determining divergence (for Mandelbrot)
- So a thread that has more black pixels to process has more work to do!

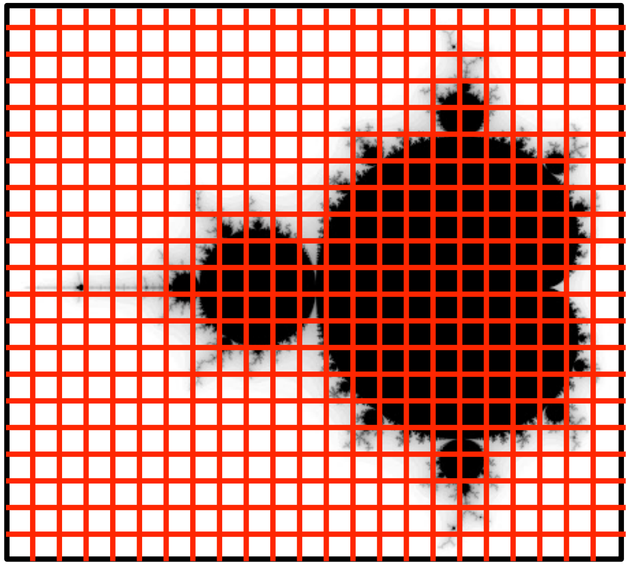


very little work

a lot of work

Better Load Balancing

- Problem: how can we achieve good load balancing across the threads?
- Key idea:
  - Decompose the domain into many "small" pieces (many more than threads)
  - Have threads compute in producer-consumer fashion



- Some "pieces" are cheap, some are expensive
- So if a thread grabs an expensive one, it won't come back for more work for a while
- In the meantime other threads can compute many small pieces

As a Concurrent Program

- We can implement this as a concurrent (not technically “parallel”) program
- We can just store the index of the next piece to be computed in a variable
  - It’s easy to have a total order of the pieces to compute (e.g., left to right, top to bottom)
- Then each time a thread is done with what it was doing (or at the very beginning), it atomically reads the index and adds one to it
  - Say we have 4 threads, they will right away grab pieces 0, 1, 2, and 3. Then whichever thread is done first will grab piece 4, and so on…
- This will be great for load-balancing, as we won’t have an “unlucky” thread that would get a lot of black pixels to compute
- It is basically a specialized producer-consumer scheme!

Load-Balancing and Overhead

- We now have a choice to make: how big/small should the pieces be?
- If we make tons of tiny pieces:
  - Great for load-balancing
  - But high overhead (i.e., threads enter the critical section a lot)
- If we make a few large pieces:
  - Great for overhead
  - Bad for load-balancing (i.e., one threads could be “unlucky” and finish well after the others)
- Depending on the use case, one should use differently sized pieces
  - But very small (i.e., one pixel) or very large (i.e., a quarter of the pixels) is likely always a bad idea

Let’s put this in practice

- All the image transformation in our app are sequential and our image app does only task parallelism
- This is great, but not always sufficient
  - Think of one large image with an expensive filter!
- So let’s add a new filter to our app and make it data-parallel!

- Let’s look at Homework #9…

Quantifying Parallel Performance

- Achieving good parallel performance is not easy
- But we should have simple metrics to quantifying it
- There are two key metrics: Parallel Speedup and Parallel Efficiency
  - Speedup: the acceleration compared to a 1-core execution
  - Parallel Efficiency: how much bang (i.e., speedup) you get for your buck (i.e., cores)
- Let's define these precisely…

Parallel Speedup

- Let T(n) be the execution time with n cores
- S(n), the parallel speedup achieved when running on n cores, is defined as:

$$S(n) = \frac{T(1)}{T(n)}$$

- Very simple metric that takes a value between 1 (no speedup!) and n (perfect, linear speedup)
- Typically we experience sublinear speedup, i.e., $S(n) < n$
  - e.g., we rarely go 10 times faster with 10 cores

Parallel Efficiency

- A high speedup is good, but we need to quantify how far it is from being ideal
- Here comes in Parallel Efficiency, E(n), defined as:

$$\boxed{E(n) = \frac{S(n)}{n}}$$

- E(n) has value between 0 and 1 (often seen as a percentage)
- Example: If with 10 cores the speedup is 4, then E(10) = 0.4 (or 40%)
  - This is means I am “wasting” 60% of my cores
  - If I didn’t, the speedup would be 10 and the efficiency would be 100%

In-Class Exercise #1

- Consider a parallel program that runs in 1 hour on a single core of a computer. The program’s execution on 6 cores has 80% parallel efficiency. What is the program’s execution time when running on 6 cores?

In-Class Exercise #1 (Solution)

- Consider a parallel program that runs in 1 hour on a single core of a computer. The program's execution on 6 cores has 80% parallel efficiency. What is the program's execution time when running on 6 cores?

- E(6) = S(6) / 6 = 0.8
- Therefore, S(6) = 4.8
- Therefore, T(1) / T(6) = 4.8
- Since T(1) = 1 hour, T(6) = 1/4.8 hours (~0.20 hours, or 12.5 minutes)

In-Class Exercise #2

- A parallel program has a speedup of 1.6 when running on 2 cores, and runs 10 minutes faster when running on 3 cores than when running on 2 cores. Give a formula for T(1) as a function of T(3)

In-Class Exercise #2 (Solution)

- A parallel program has a speedup of 1.6 when running on 2 cores, and runs 10 minutes faster when running on 3 cores than when running on 2 cores. Give a formula for T(1) as a function of T(3)
- T(1) / T(2) = 1.6
- T(3) = T(2) - 10
- So T(3) = T(1)/1.6 - 10
- meaning that T(1) = 1.6 * (T(3) + 10)

Exposing Data Parallelism

- What we often need to do, and what you'll do in Homework #9, is to "expose" data parallelism
  - i.e., identify which part of the code can be made data parallel
- In our homework assignment it's trivial because our image filter is very simple
- But it's not always the case that the entire code can be made data-parallel
  - And in fact, in our app, the I/O is not parallelized
- So often we are faced with situations in which we have to leave part of the code unparallelized
- The longer is spent in the non-parallelized part of the execution, the worse it is to parallel speedup and parallel efficiency

EduWrench Module

- You may have taken a course from me in the past in which we used simulation
- Based on those (I think, successful) experiences, I did received funding to create more simulation-driven pedagogic content
- All material is at https://eduwrench.org
  - Feel free to browse that site
- For now, let’s use it to learn our last key data-parallelism concept…

Data Parallelism and Amdahl’s Law

- Let’s do the following:
  - We all go to http://eduwrench.org right now
  - Sign in using our @hawaii.edu account
  - Go to: MODULES::Multi-Core Computing and click on the Data Parallelism tab
- Then:
  - I go through some of the intro material with you
  - You then use the simulation to answer three practice questions
  - I then go through the Amdahl’s Law content
- And Then:
  - At home, you review this content and go through the remaining content and do practice questions on your own
  - You then do a short pencil and paper Homework Assignment

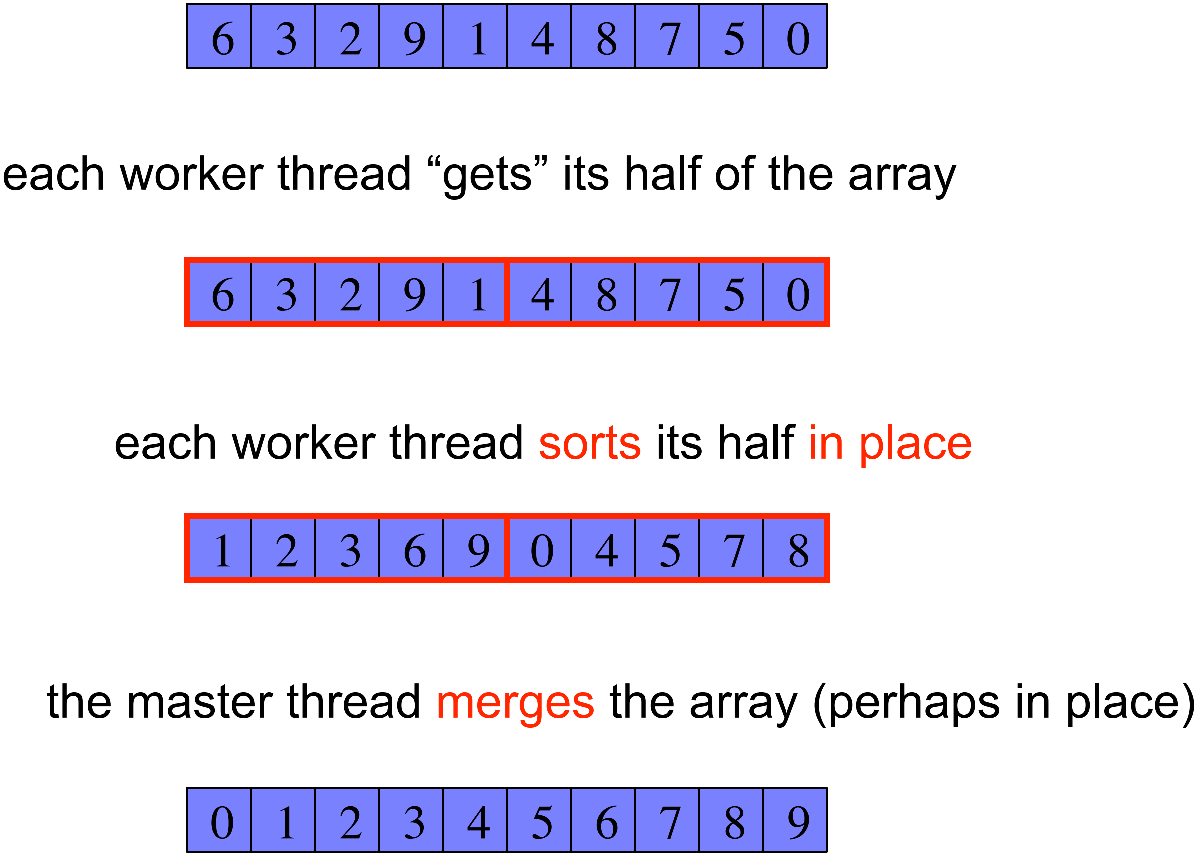
What about Sorting?

- In an Algorithms course you learn about sorting
- What about multi-threaded sorting?

Sorting an Array with Threads

- Consider an array of n elements to sort
- Let's say you have a machine with 2 cores
- One approach is to split the array in two among two threads
  - Each sorting can be done in O(n log n)
  - Then merging is in O(n)
  - Therefore, if the array is large, on should get close to a speedup of 2 because the sorting (which is done in parallel) is the dominant operation
    - But we know by Amdahl law that for non-huge arrays we could really be hurt by the sequential merge
    - And a log n factor isn't a lot
- Note that we do not need any mutual exclusion here, because we're sorting disjoint pieces of the array
  - This is typically called "parallel" computing rather than "concurrent" computing

Sorting with Threads



6	3	2	9	1	4	8	7	5	0

each worker thread “gets” its half of the array

6	3	2	9	1	4	8	7	5	0

each worker thread sorts its half in place

1	2	3	6	9	0	4	5	7	8

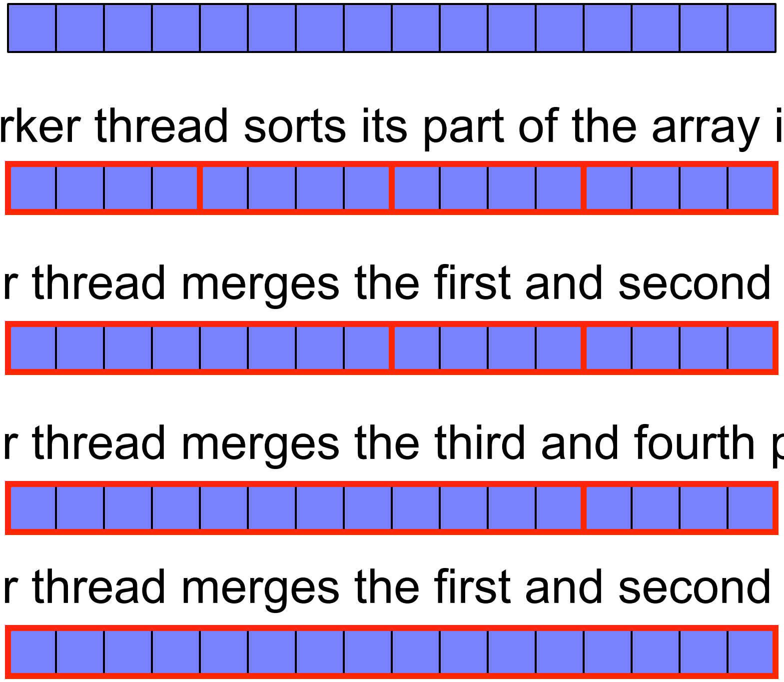
the master thread merges the array (perhaps in place)

0	1	2	3	4	5	6	7	8	9

What about using more threads?

- What about using more threads to exploit more processors/cores?
- One possibility: cut the array in T pieces, where T is the number of threads
- Drawbacks:
  - Merging becomes more complicated
  - And it has higher complexity

Using 4 threads



each worker thread sorts its part of the array in place

master thread merges the first and second piece

master thread merges the third and fourth piece

master thread merges the first and second piece

done

Any hope for parallel performance?

- Let n be the size of the array, and p the number of processors
  - Assume p divides n
- The complexity of the merging is approximately O(n log n), which is not good
- Amdahl’s law tells us that even a small sequential part can be bad
- And in this case it may not even be that small at all
- So let’s parallelize it!

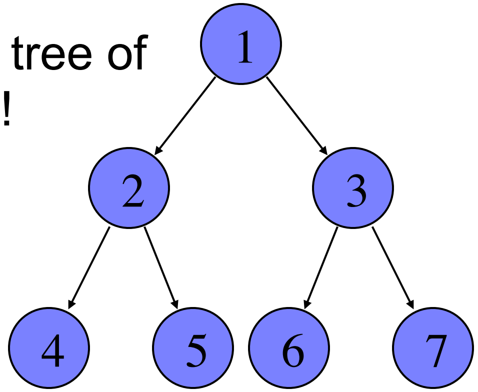
Multi-threaded merging?

- One solution is to write a multi-threaded merge routine that does the merges in parallel
  - takes as input A, n, and p.
  - uses p threads
- This is not very elegant because
  - One creates p threads to do the sorting
  - We wait until everything is sorted
  - We terminate the p threads
  - We create p new threads to do the merging
- A more elegant implementation is to do the partial sorting and partial merging all at the same time recursively

Recursive multi-threading

- Create a function that does the sorting of one array by
  - creating two threads to do partial sorting
  - doing the merging
- The threads doing the partial sorting call this function, and thus can create threads themselves

a binary tree of threads!!



Implementations

- The course web site points to a Makefile and several implementations in C using Pthreads:
    - Sequential
    - Parallel sort and sequential merge
    - Parallel sort and parallel merge
    - Recursive parallel “sort and merge”
- Let’s look at the code and run the Makefile….

Sorting Performance?

- More threads is good
  - The more threads the better we can use multiple cores
- More threads is bad:
  - The more threads the more merging operations
    - But merging happens hopefully concurrently
  - The more threads the more “thread overhead”
- What about Load Balancing?
  - It is possible that the left branch of the tree, i.e., the left half of the array is more difficult to sort than the right half
  - But since many threads are created recursively, as long as we have P threads we can keep a P-core machine busy
  - Therefore more threads is good:
- The number of threads is controlled by the depth of the tree, and in our case by the “base case size”
- There is probably a best “base case size”, which should be determined experimentally

Parallel Sorting is not Easy

- As we know, a common performance bottleneck is the memory
- The more computation the better, i.e., the higher the computational complexity the better
- Parallelizing an O(n) computation with O(n) memory accesses can only yield minor benefits
  - Unless the constant hidden in the O is large
- Parallelizing a $O(n^5)$ computation should be “easier”, in the sense that there should be more opportunity to utilize the core’s computing power without being killed by the memory bottleneck
- Efficient parallel sorting is actually a well recognized difficult problem with a large literature

Conclusion

- Data parallelism can be applied to many applications, and in particular stencil applications
- Achieving good data-parallelism performance on multi-core machines is not alway easy
  - e.g., tension between overhead and load-balancing
- GPUs are really good at data parallelism

- We already looked at Homework #9
- Let’s look at Homework #10…