Concepts in parallel programming

Warwick RSE

15/12/2023

Task parallelism

Task parallelism

- Split up problem into separate tasks
  - Explore parameter space
  - Multiple Monte-Carlo realisations
- Run each task on a separate processor of your computer
  - Or separate computers

Task parallelism

- Scales very well
    - Only limit on scaling is number of tasks and number of processors
    - If you have one processor available per task then you can get all of your tasks done in the same time as one task
    - Slight wrinkle if each task is so quick that it takes as long to start the task as it does for it to run

GNU Parallel

- Installed on the clusters and the COW (including dedicated nodes)
  - Will need the module “parallel” on the cluster systems
- Can be installed easily on most other *nix platforms
  - sudo apt-get install parallel
  - yum install parallel
  - brew install parallel

GNU Parallel

- Very good official tutorial at https://www.gnu.org/software/parallel/parallel_tutorial.html

- Idea is that you create a program that takes parameters through the command line and then tell parallel how to build command lines to run several copies at the same time
    - Put together program name, parameters etc.
- Has a number of job slots (usually the number of processors that you have)
    - Runs tasks in sequence on each job slot until it runs out of task

GNU Parallel

```
#!/bin/bash

parallel echo ::: A B C D E F
```

- Very simple Parallel script
- “echo” is a command line utility that just prints its arguments

```
A
B
C
D
E
F
```

GNU Parallel

```
#!/bin/bash

parallel echo ::: A B C D E F
```

- “parallel” just runs the GNU parallel program
- “echo” here just prints the parameters that it is given but in general would be the program that you want to run in parallel (including any command line arguments that don’t change)
- “:::” separates the command that you want to run from the parameters that you want to run it with
- The last bit is the list of parameters that you want to pass to your program separated by spaces (you can change the separator if you want)

GNU Parallel

```
#!/bin/bash

parallel echo ::: A B C ::: D E F ::: G H I
```

- Putting in multiple argument sources calls the program with every possible combination of the sources (Cartesian Product)

```
A D G
A D H
A D I
A E G
A E H
A E I
A F G
A F H
A F I
B D G
B D H
B D I
…
…
…
```

GNU Parallel

```
#!/bin/bash

parallel echo HELLO {} ::: A B C
```

- You can specify where to put the command line argument that Parallel generates using {}

```
HELLO A
HELLO B
HELLO C
```

GNU Parallel

```
#!/bin/bash

parallel echo {1} Says hello to {2} ::: A B C ::: D E F
```

- If you use multiple argument sources then you can choose which one to use using {number}

```
A Says hello to D
A Says hello to E
A Says hello to F
B Says hello to D
B Says hello to E
B Says hello to F
C Says hello to D
C Says hello to E
C Says hello to F
```

GNU Parallel

```
#!/bin/bash

parallel echo {1} is job number {#} ::: A B C
```

- You can get the job number for each parallel job using {#}. The job number goes up from 1 as each job is run

```
A is job number 1
B is job number 2
C is job number 3
```

GNU Parallel

```
#!/bin/bash

runfunc(){
  echo $1 has uuid `uuidgen`
}

export -f runfunc
parallel runfunc ::: A B C
```

- If you need a unique ID then you can use the uuidgen command line program to generate one

- Your UUIDs will be different to these

```
A has uuid 536b2247-64fb-49cd-af9b-258ca52c6200
B has uuid fef2fce0-26fb-4553-9c5e-7a6c8b07cd58
C has uuid bc58cea0-8ffb-4733-9650-441f3587023d
```

GNU Parallel

- You can use Parallel to run
  - Any shell command (ls, cat, echo etc.)
  - Any program that can be run from the command line
  - bash functions if they are exported using “export -f”

GNU Parallel

- Lots of other options but you can already see how you can use Parallel to run a wide variety of jobs

- Details for our cluster
  - https://wiki.csc.warwick.ac.uk/twiki/bin/view/HPC/ClusterUserGuide#Serial_jobs

- You can easily make pretty much any modern language take command line parameters
  - Can combine with input files (specified by name) for more sophisticated control

MapReduce

- MapReduce is a method of processing large amounts of data in parallel

- It consists of two core operations (there are more in a real system), one runs independently on each processor, one runs between processors

  - Map - Convert raw data to a quantity that you want to process. This happens completely independently on each processor

  - Reduce - Operate on two mapped data items to produce a new mapped data item containing the data from both original data item (linear, commutative and associative combination)

MapReduce

- The system works because the reduction function combines two items into another item of the same kind

- Two of these new item can then be combined as well

- So the system can map and reduce all of the data on a single processor, take the reduced data from that processor and combine it with reduced data from other processors

    - It can be complex to work out how to do this reduction fast on real hardware but this is the idea

MapReduce

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

MapReduce

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

MapReduce

It was the best of times,	it was the worst of times,	it was the age of wisdom,	it was the age of foolishness
it = 1 was = 1 the = 1 best = 1 of = 1 times =1	it = 1 was = 1 the = 1 worst = 1 of = 1 times =1	it = 1 was = 1 the = 1 age = 1 of = 1 wisdom =1	it = 1 was = 1 the = 1 age = 1 of = 1 foolishness =1

MapReduce

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

it = 1 was = 1 the = 1 best = 1 of = 1 times =1	it = 1 was = 1 the = 1 worst = 1 of = 1 times =1	it = 1 was = 1 the = 1 age = 1 of = 1 wisdom =1	it = 1 was = 1 the = 1 age = 1 of = 1 foolishness =1

it = 2 was = 2 the = 2 best = 1 worst = 1 of = 2 times =2	it = 2 was = 2 the = 2 age = 2 of = 2 wisdom =1 foolishness = 1

MapReduce

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

it = 2
was = 2
the = 2
best = 1
worst = 1
of = 2
times =2

it = 2
was = 2
the = 2
age = 2
of = 2
wisdom =1
foolishness = 1

MapReduce

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness

it = 2
was = 2
the = 2
best = 1
worst = 1
of = 2
times = 2

it = 2
was = 2
the = 2
age = 2
of = 2
wisdom =1
foolishness = 1

it = 4
was = 4
the = 4
best = 1
worst = 1
of = 4
times = 2
age = 2
wisdom = 1
foolishness = 1

MapReduce

- There are some substantial restrictions on what you can do with map reduce

- Most important is that you have to be able to combine operations on already partly reduced data

- So averaging doesn’t work directly. If you split a set of numbers into two groups then the average of all the numbers is not the average of the average of the two groups

    - You can do an average because the sum of the numbers does combine as required. You have to divide by the total number at the end

MapReduce

- There are a lot of packages that implement MapReduce type semantics
  - Apache Hadoop
  - MongoDB
- There are several front ends to such packages in R or Python (although a lot of them do require you to write other languages. MongoDB uses Javascript for writing the map and reduce functions)
- MapReduce models are the simplest of the “Big Data” processing systems. There are more sophisticated ones but MapReduce is a good place to start

MapReduce

- The trick with MapReduce is to work out how to do as much of the work in the map (fully parallel) section and as little as possible in the reduce (partly parallel)

- This can involve changing algorithm or using the algorithm differently to how you would on a single processor

  - More parallelism > More speed

Connected Task Parallelism

- GNU Parallel is brilliant when you have a set of problems to run that are independent

- What if your problems are not independent?

  - You can use a worker-controller model instead

- For most of the period of computer programming this model was called “Master-Slave” but some people prefer a different term now

  - There is no agreement on a replacement but we will use worker-controller

Forensic Linguistics

- Wikipedia lists nearly 100 people who might have actually written the plays attributed to William Shakespeare. Most likely it was who everyone thought it was

- Imagine that you wanted to write a program to test if an author could be responsible for the works of Shakespeare

- Some tests are easy (was the person alive at the time?) but if they flag false then they mean you don't need to do the harder ones

- The hardest tests (machine learning analysis of sentence structure?) can take 1000s of times longer to run than the simplest

DNA Sequencing

- Imagine that you want to test a known sequence of DNA to see if a sequence of incomplete DNA "chunks" came from decomposition of that sequence

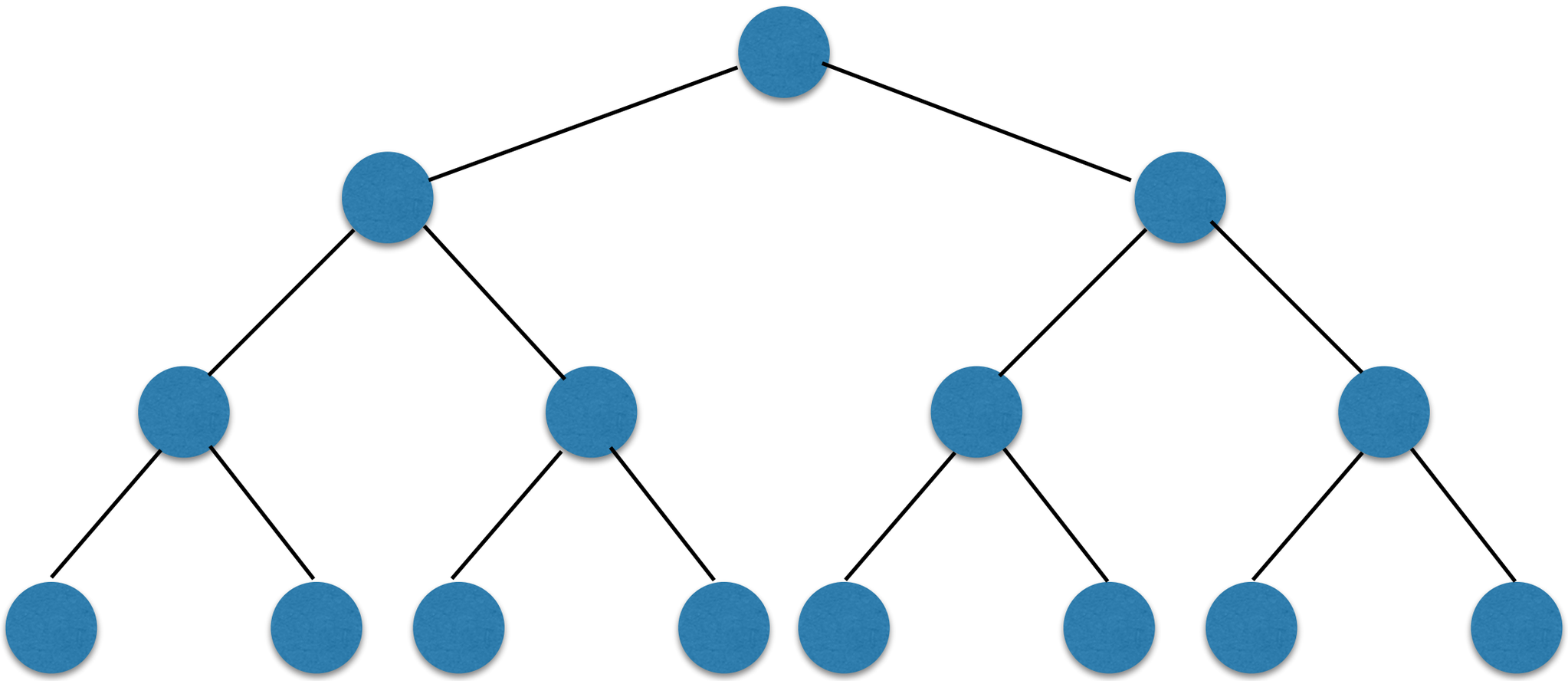
- Reconstituting sequences of DNA from chunks is (comparatively) expensive so you want to construct the smallest sequences that let you know if the sequence came from the candidate (see shotgun sequencing)

- You want to reject candidates from the shortest chunk that is possible

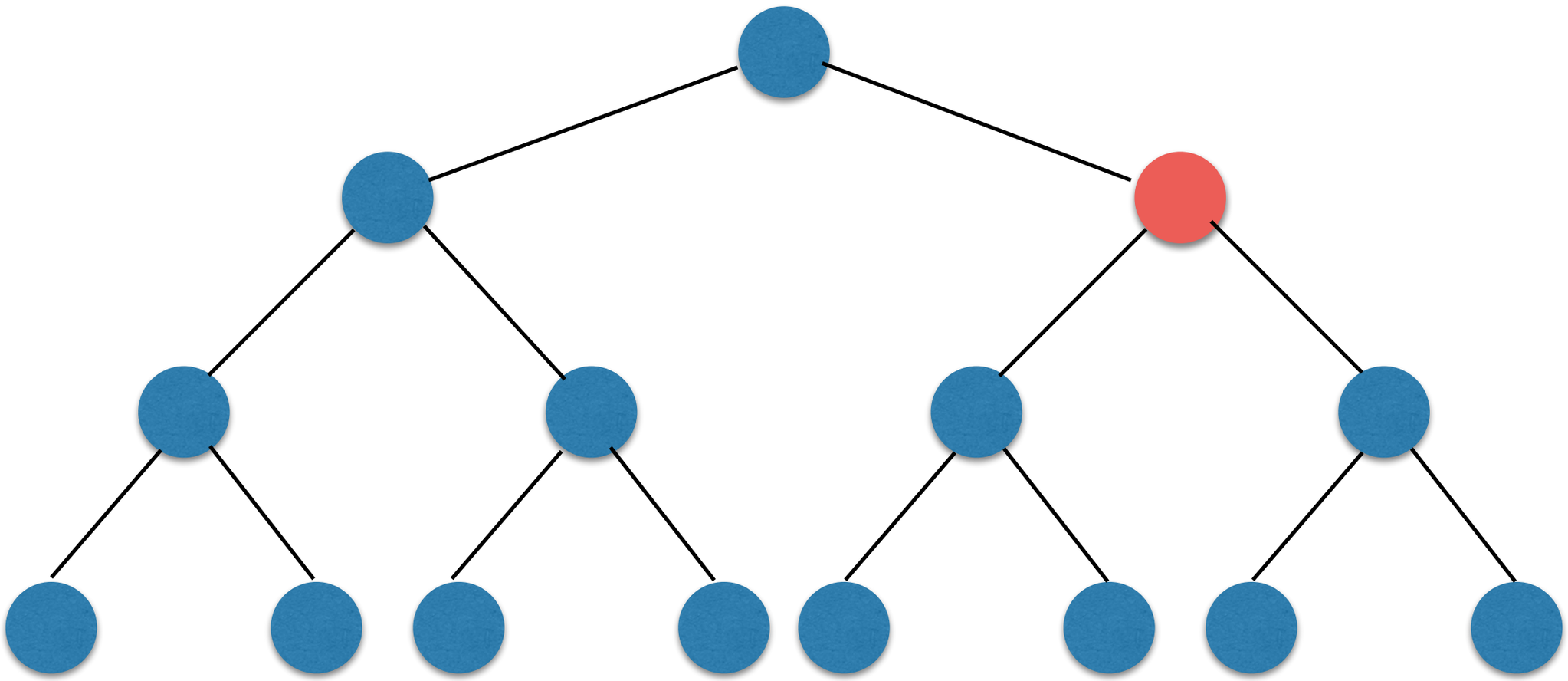
    - Reconstruction and matching are not perfect so you have to run until you have a given level of certainty of match/not match

        - Also want to reject reconstruction if not present in any candidates (or at least consider probabilities)

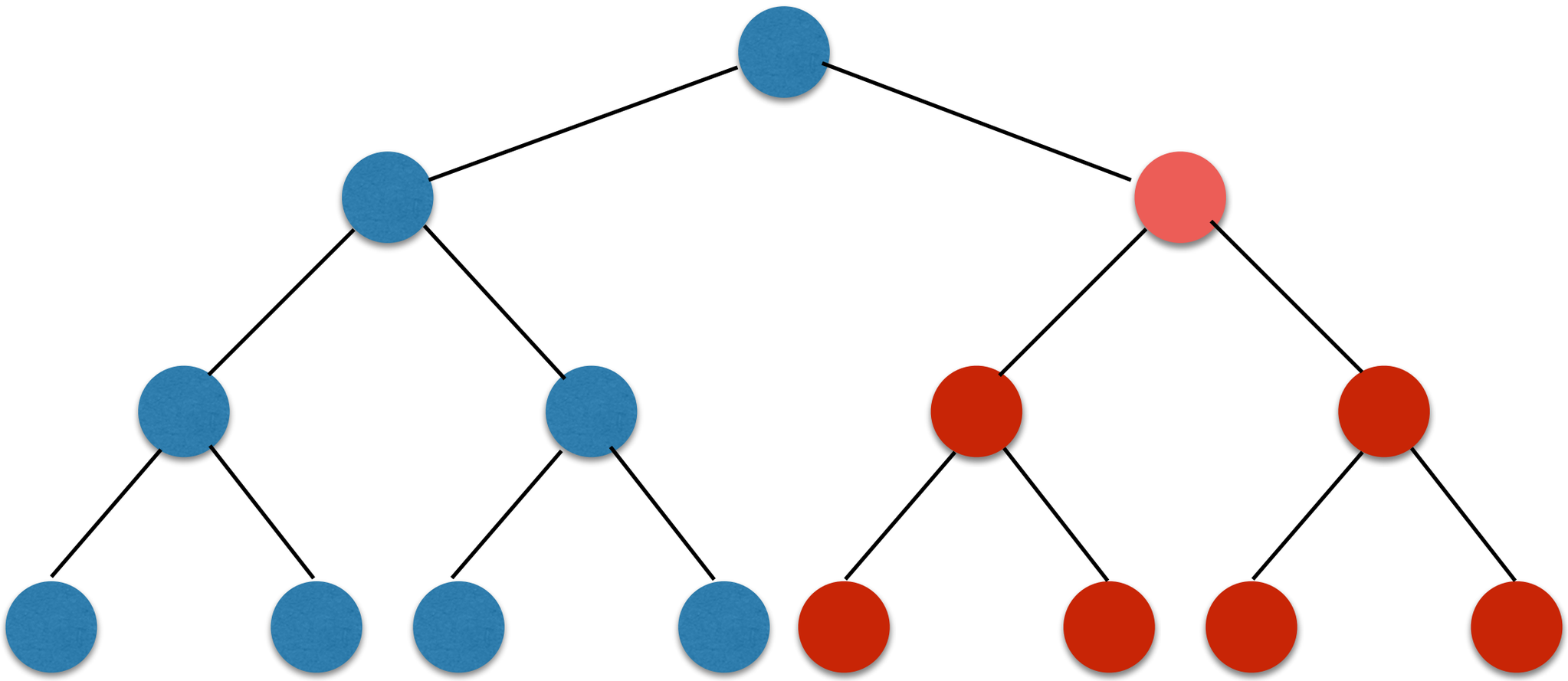
Tree Pruning



Tree Pruning



Tree Pruning



Only test HALF of possibilities

Worker-Controller

- The idea is that you have one processor (the controller) that dispatches packages of work to the other (worker) processors
  - Unlike with Parallel the controller can make choices about which work package to hand out
  - If a result indicates that some work packages are no longer needed then it can choose not to hand them out to workers
- Problem is that there isn't any standard package to let you set up this paradigm

Worker-Controller

- Have to have communication between the controller and the worker
  - Network
    - OS parallel processing feature
    - Files
- Can work using files easily enough but performance is poor
- Network communication and OS features are generally best used through a library or framework

Worker-Controller

- Can implement the paradigm in pretty much any parallel system
  - We'll describe the options later
- Some systems have built in support for it
- Some libraries are intended for this type of operation

Worker-Controller

- C/C++/Fortran
    - MPI
    - OpenMP
    - Threading
    - Low level networking (**very** unusual in academia)
    - CUDA

Worker-Controller

- Python
  - Python Multiprocessing
  - Dask (https://dask.org/) (Dask is huge and does do other things too)
  - Ray (https://ray.readthedocs.io/en/latest/)
  - MPI4Py
  - Lots of others (Faust, gevent, Thespian)
- Always be careful when using parallel programming in Python. Your core code can sometimes be slow enough that you would get better performance rewriting it in C/C++/Fortran

Dask

- Python library for parallelism
- Dask is primarily aimed at distributed analytics
  - You use Dask replacements for things like Pandas dataframes (and lots of other things) that work in parallel
  - Both on a single machine and on a cluster
  - Exactly how these work depends on what Dask is replacing
- There is also a generic approach in Dask called “futures”

Dask

- The idea of a future is to say “I want you to run this problem on this data” for multiple problems and/or multiple pieces of data and let the system sort out how to actually run them

- Dask futures are the same idea as Python concurrent futures but can run on clusters

- You only worry about when and where they are running when you need the results.

Example on Warwick's Sulis

```
# Query SLURM environment per worker task
p = int(os.getenv('SLURM_CPUS_PER_TASK'))
mem = os.getenv('SLURM_MEM_PER_CPU')
mem = str(int(mem)*p)+'MB'

# Initialise Dask cluster and client interface
dask_mpi.initialize(interface = 'ib0', nthreads=p, local_directory='/tmp', memory_limit=mem)
client = dask.distributed.Client()

# We expect SLURM_NTASKS-2 workers
N = int(os.getenv('SLURM_NTASKS'))-2

# Wait for these workers and report
client.wait_for_workers(n_workers=N)

num_workers = len(client.scheduler_info()['workers'])
print("%d workers available and ready"%num_workers)

# Create a list of inputs to the work function
inputs = range(N)

# Evaluate f for all inputs using a pool of processes
outputs = client.map(work_func, inputs)

print([output.result() for output in outputs])

client.shutdown()
```

Dask

- Three phases
  - Set up Dask cluster
  - Create futures (here using client.map to map multiple inputs to the same function, but there are other ways of creating futures)
  - Wait for the futures to complete
- The power is that you can keep creating new futures etc. so you can set up new problems based on the results of previous problems

Tightly coupled parallelism

- The hardest form of parallelism to exploit is tightly coupled parallelism

- You don’t have lots of problems that you want to schedule in some way - you have one problem that you want to split up

- This often involves lots of communication between processors because you have to get information from neighbouring processors to “glue” the edges back together

Tightly coupled parallelism

- This “glue” is the important difference between tightly coupled and the previous types of parallelism

- Imagine that you are finding all of the prime numbers between 1 and 1,000,000.

- This is a single problem but is not (intrinsically) tightly coupled

- I can test every prime number separately so I can split the problem up into independent chunks

    - Actual prime finding algorithms (even simple-ish ones like the sieve of Erathosthanes) can actually be coupled but this isn’t intrinsic

Tightly coupled parallelism

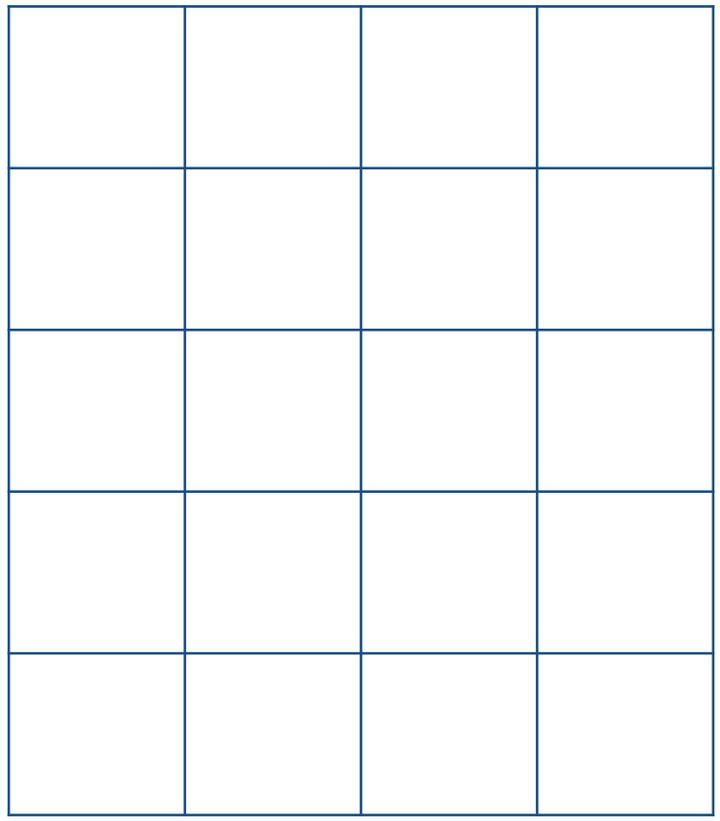
- A simple example case would be iterative smoothing of an image

- The simplest way of smoothing an image is to average each pixel with it's neighbours

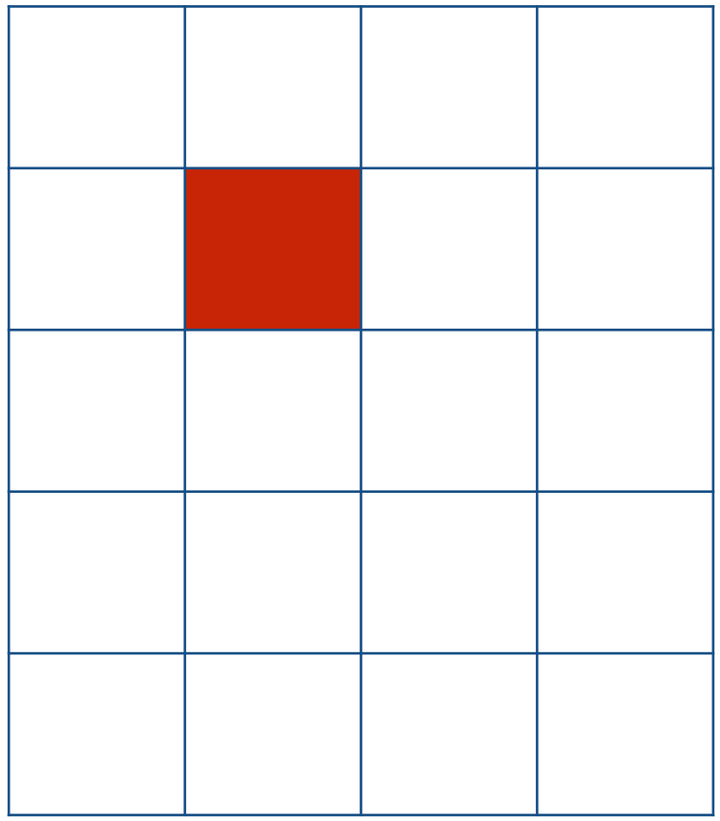
- Then keep doing it until the image is as smoothed as required

```
DO iy = 1, ny
  DO ix = 1, nx
    temp(ix,iy) = 0.25 * (im(ix-1, iy) + im(ix+1, iy) + im(ix,iy-1) + im(ix,iy+1))
  END DO
END DO
im = temp
```

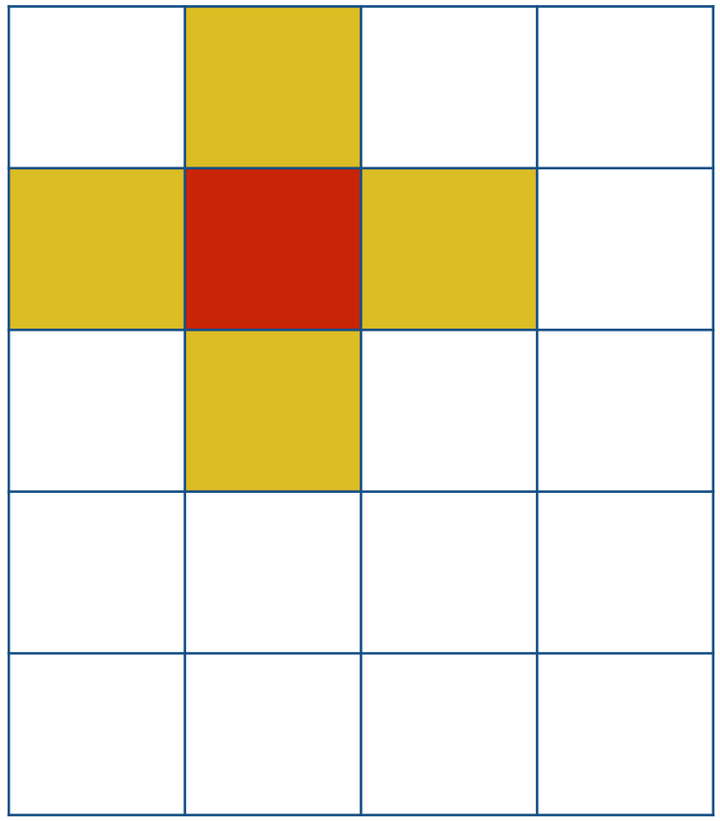
Tightly coupled parallelism



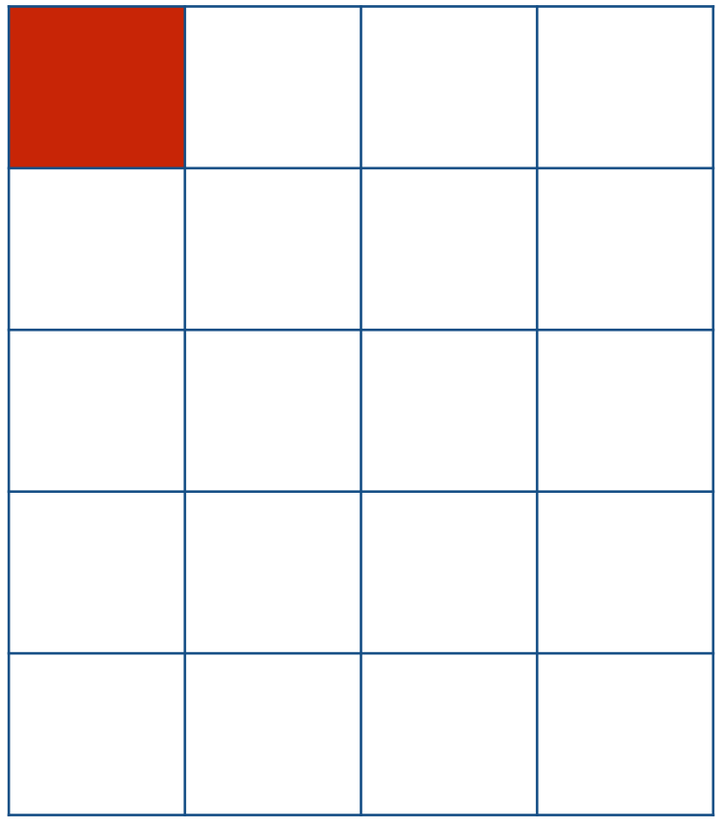
Tightly coupled parallelism



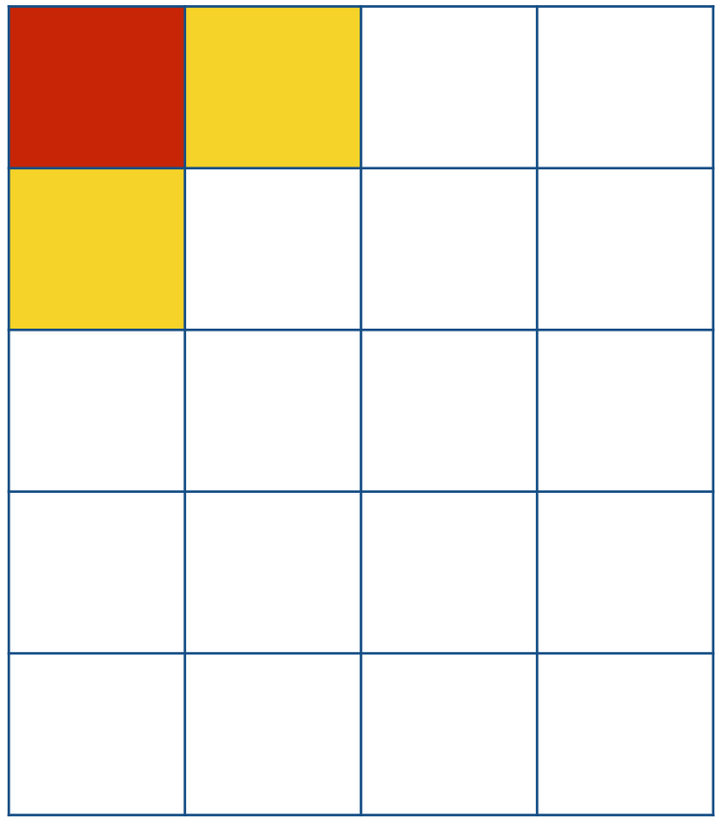
Tightly coupled parallelism



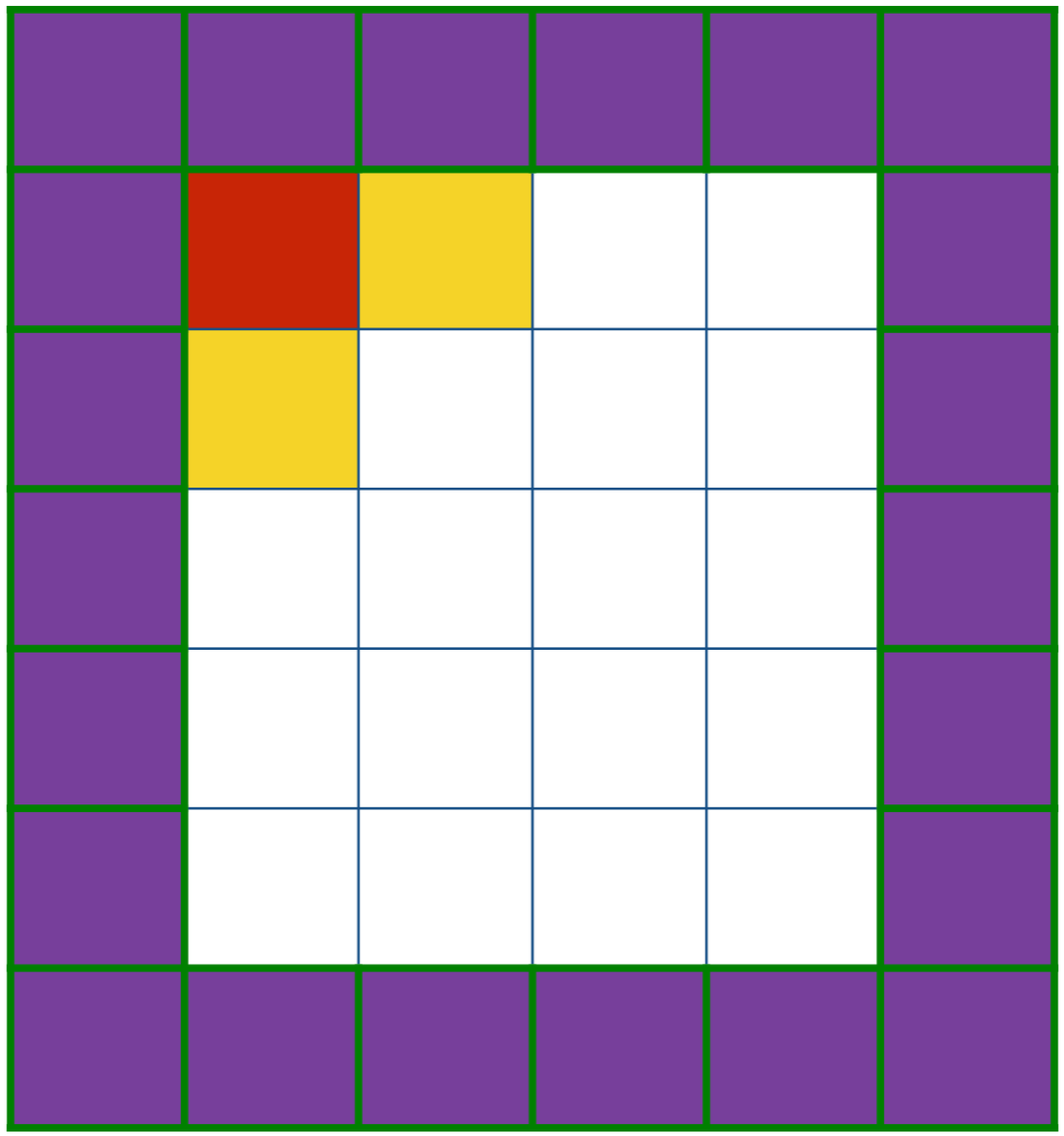
Tightly coupled parallelism



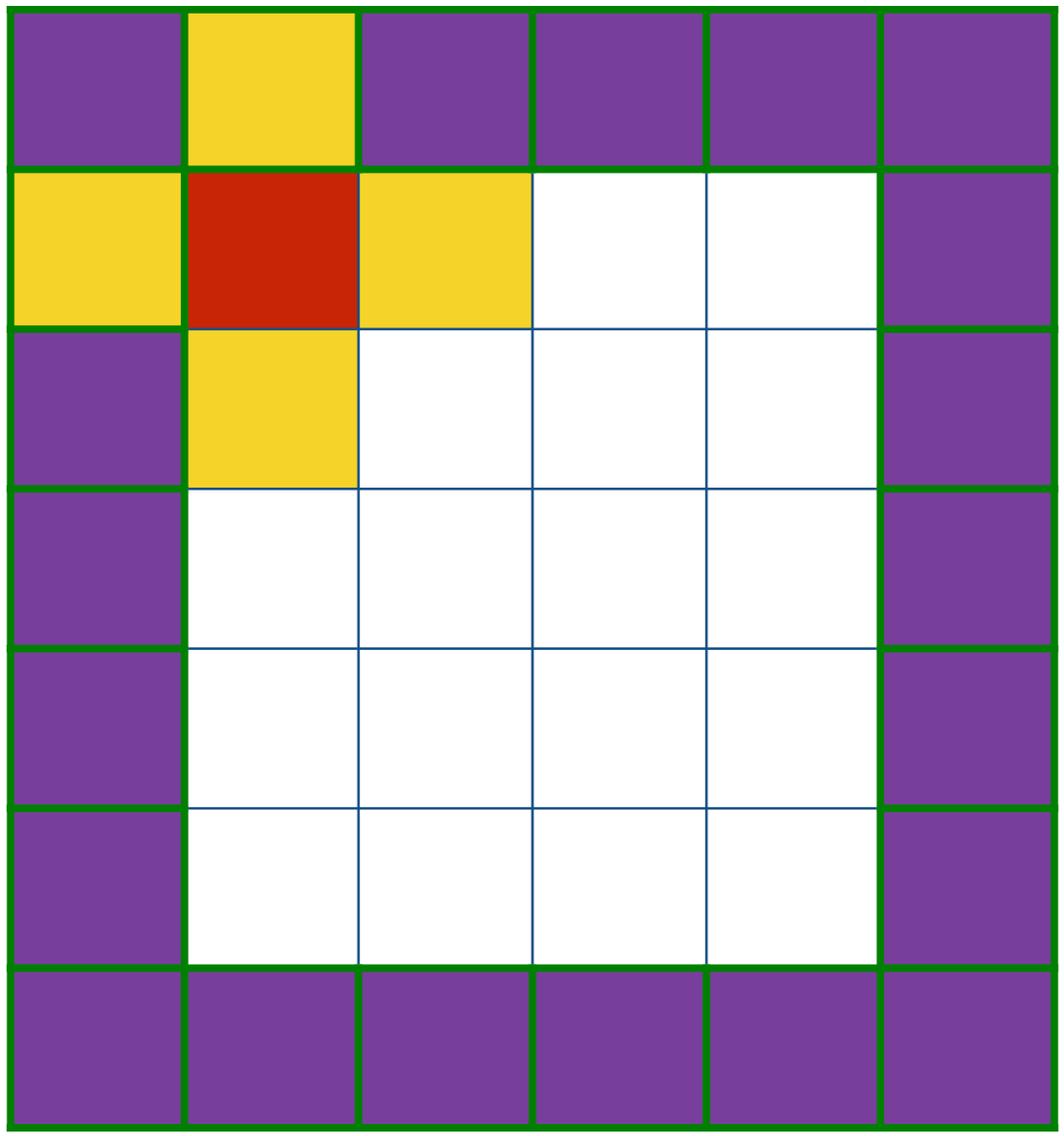
Tightly coupled parallelism



Tightly coupled parallelism



Tightly coupled parallelism



Tightly coupled parallelism

- To update every point you use points on either side of it

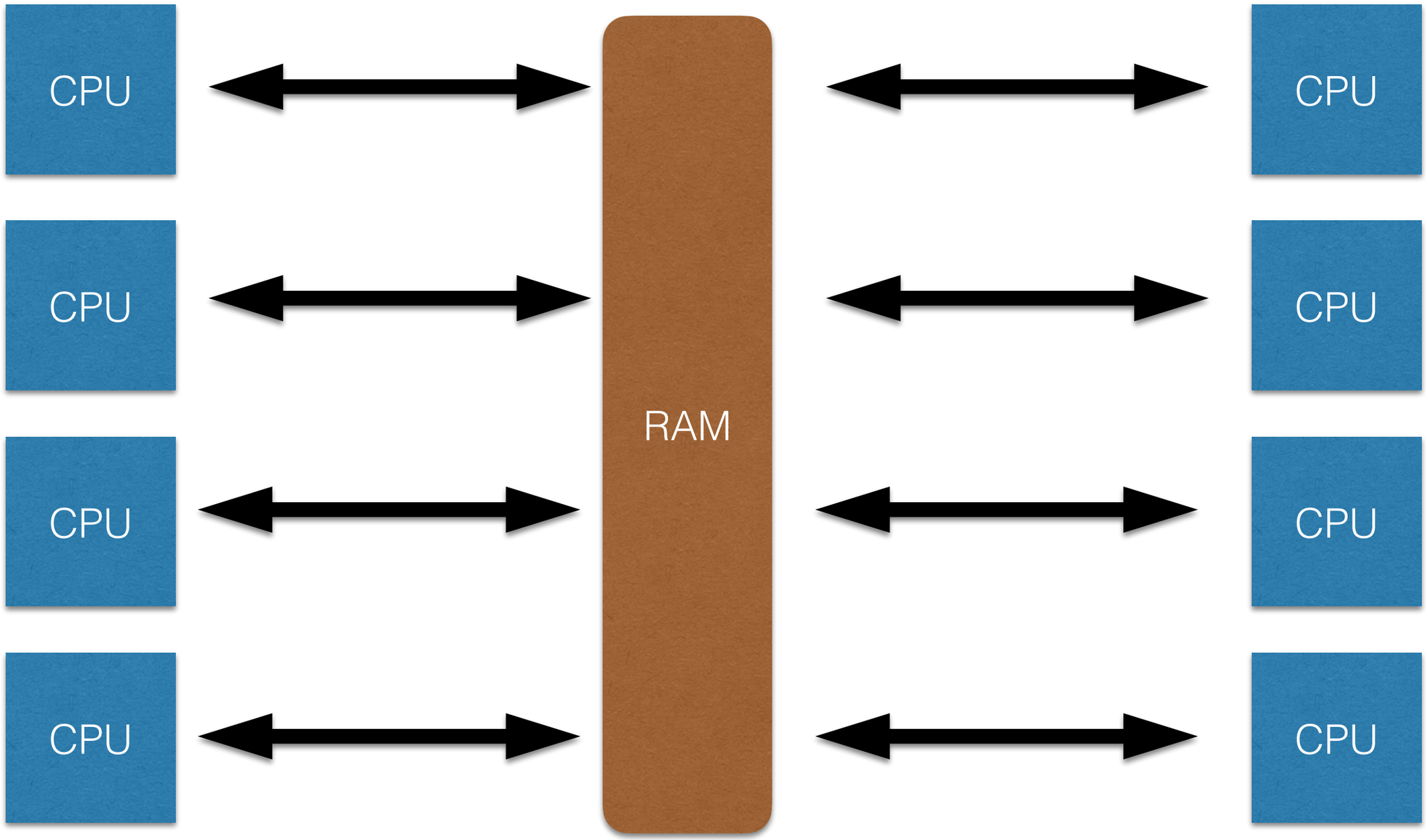
- This means that if you split your work up you're fine up until you reach the edges of the bit that you're working on then you need the data from one of the other bits

- This then splits into two

  - Shared Memory - Can just reach out and get the data

  - Distributed Memory - You can't. The other processor has to give you your data (sort of anyway)

Shared Memory



Shared Memory

- Your processors all share a single pool of memory
  - Normal multi-core computers are like this
- What you have to do is ensure that processors don't modify memory in ways that other processors don't expect
  - In the image smoothing case need to stop the copying back of the smoothed image until all processors have finished smoothing

```
DO iy = 1, ny
  DO ix = 1, nx
    temp(ix,iy) = 0.25 * (im(ix-1, iy) + im(ix+1, iy) + im(ix,iy-1) + im(ix,iy+1))
  END DO
END DO
!SYNCHRONIZE HERE
im = temp
```

Shared Memory

- There are two main approaches that are commonly used in academic programming to program for shared memory

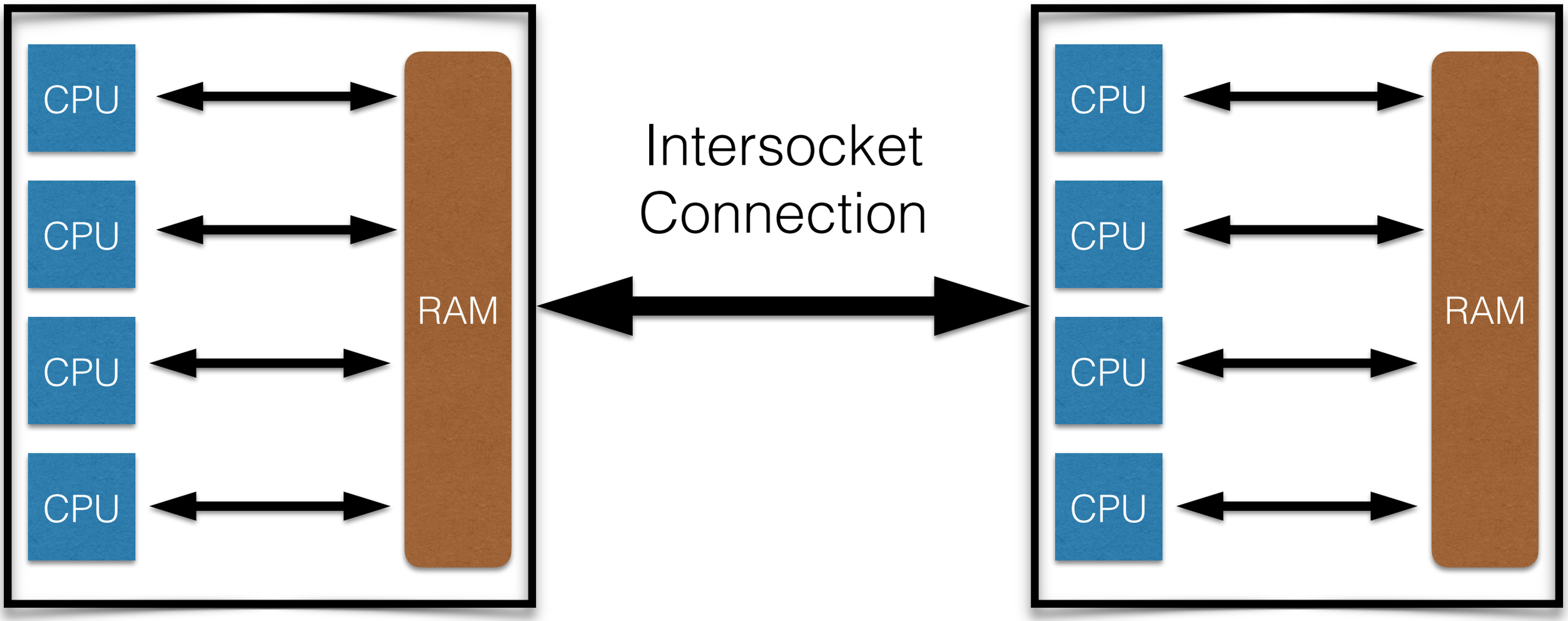
  - Threading - Essentially run a given function on a separate processor (in a separate **thread**). Several different methods to write threaded code. Core library on *nix is called pthreads

  - OpenMP - Versatile library but the core and most common form is to instruct the compiler to split up loops to run on separate processors. Only available for C/C++/Fortran

Shared Memory

- Plenty of other libraries
  - Intel Threading Building Blocks is a C++ template library that gives you features similar to OpenMP
    - Part of a larger library called OneAPI that covers a lot of parallelism
    - OpenAcc tries to allow parallel programming on CPUs or GPUs (although this is also available in OpenMP, it's more mature in OpenAcc)
- There are also some languages aimed at parallel programming
  - Be careful! There are a lot of languages of this type that have been discontinued

Non-Uniform Memory Access



$4GHz \sim 2.5 \times 10^{-10}s$

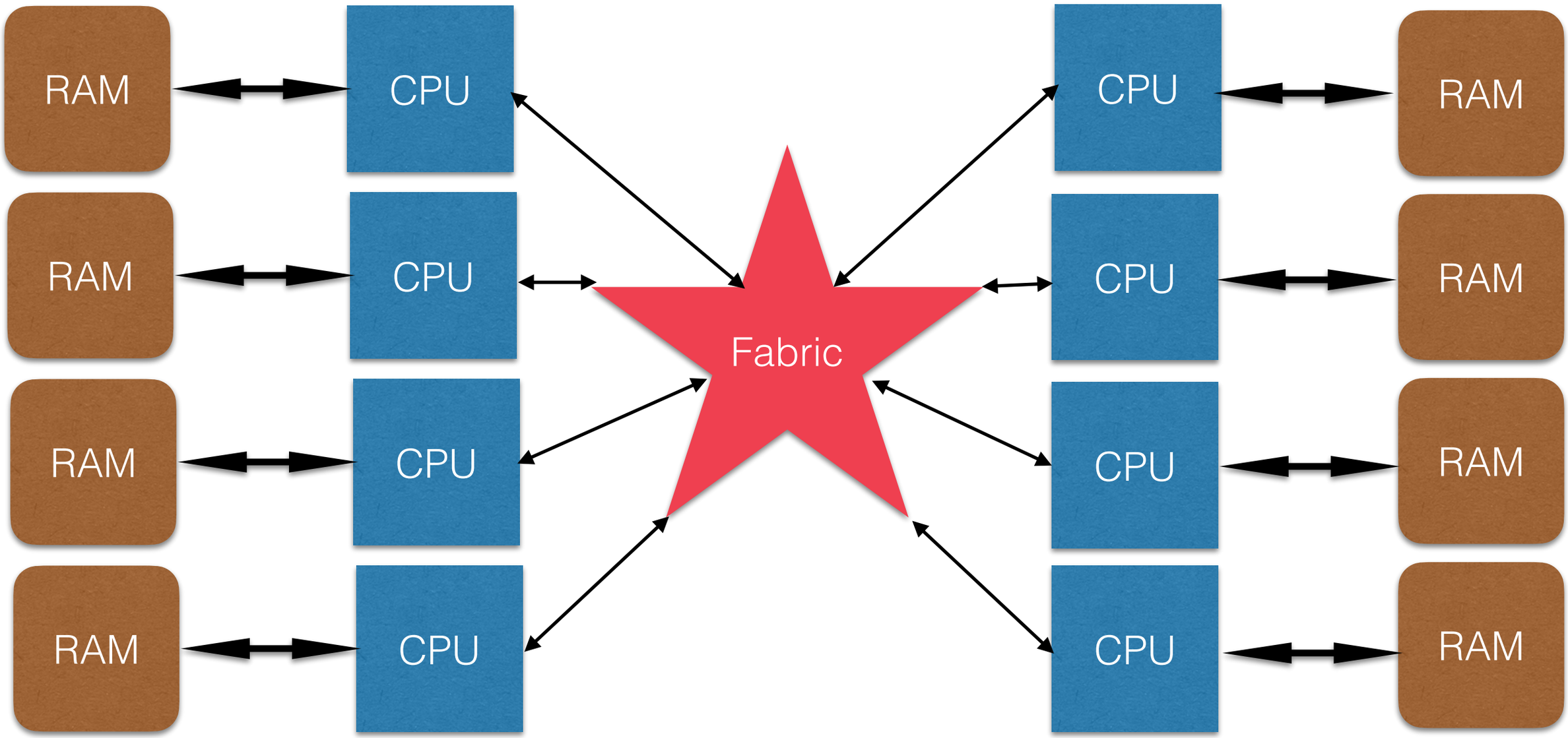
$2.5 \times 10^{-10}\,s * c = 7.5cm$

Data on separate processor **must** be slower to access

NUMA

- Mostly you as a programmer don't have to worry about NUMA
  - From a programming perspective it is identical to shared memory
  - Some memory is slower to access than other memory
    - In particular the first time a variable is set the memory for it will be put in the memory of the processor that set it
    - Worrying about NUMA is a (usually about 5% ish) performance improvement not a programming problem
      - Performance benefit is larger on really big CPUs like the newer 50-128 core chips

Distributed Memory



Distributed Memory

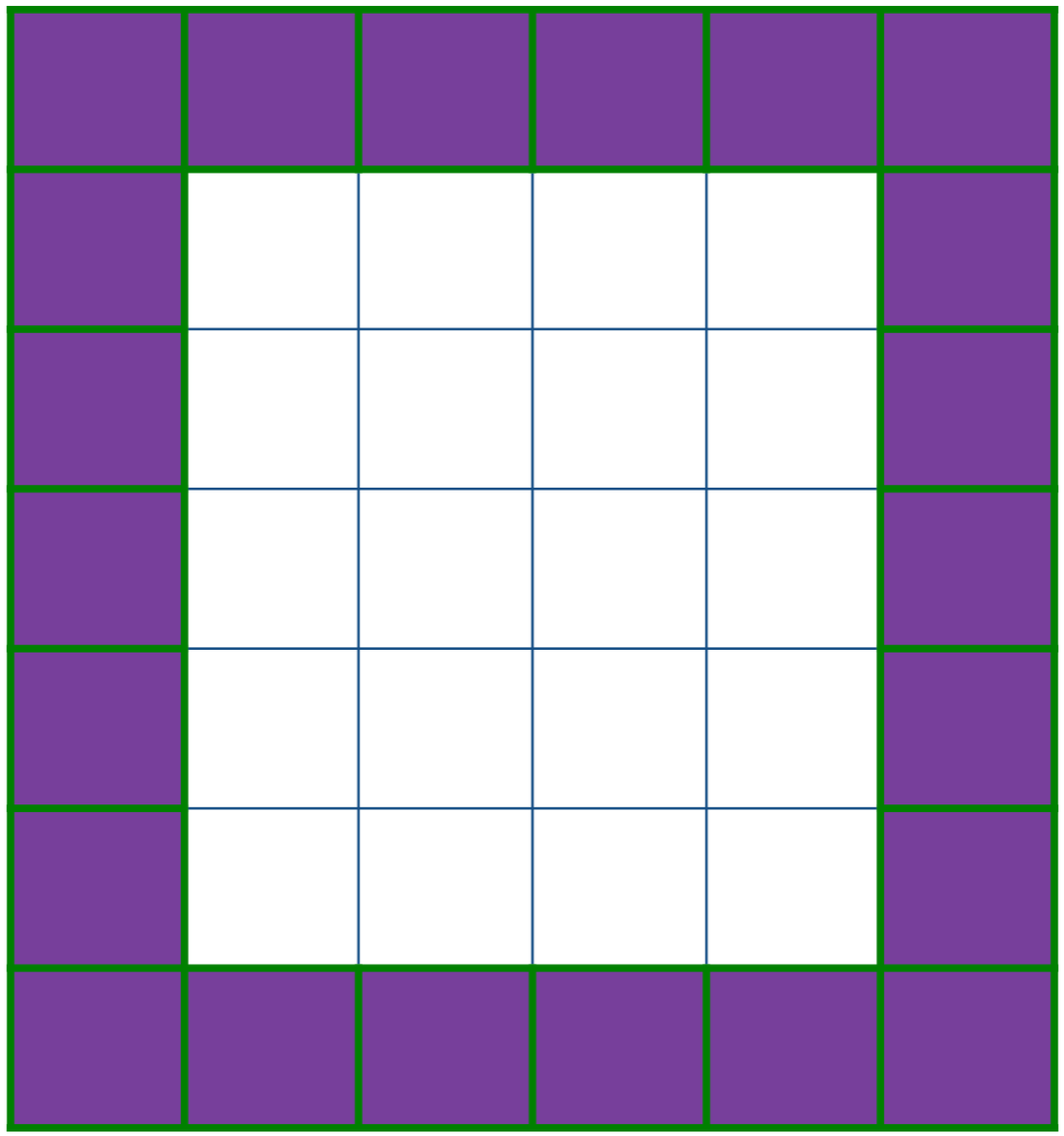
- Your processors have their own memory and have to send information from one to the other to know what data each has

- Because you now have control of when data is sent between processors you don't have to worry about synchronisation as much

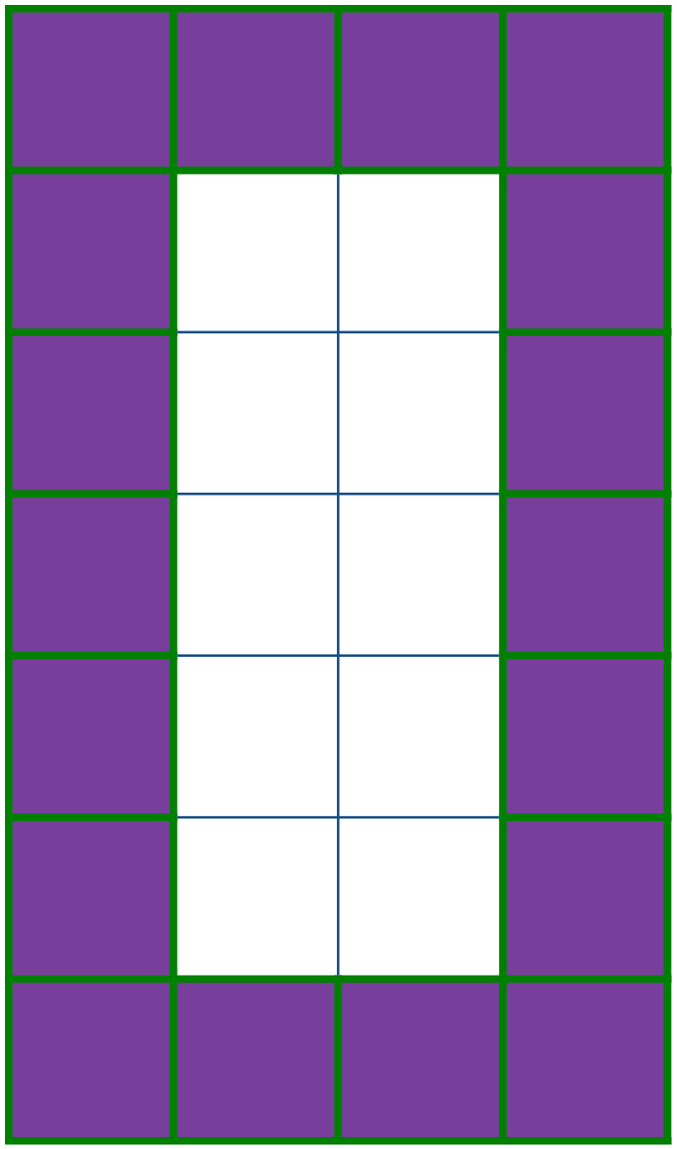
- The sender chooses when to send the data so it know that it is in the right state

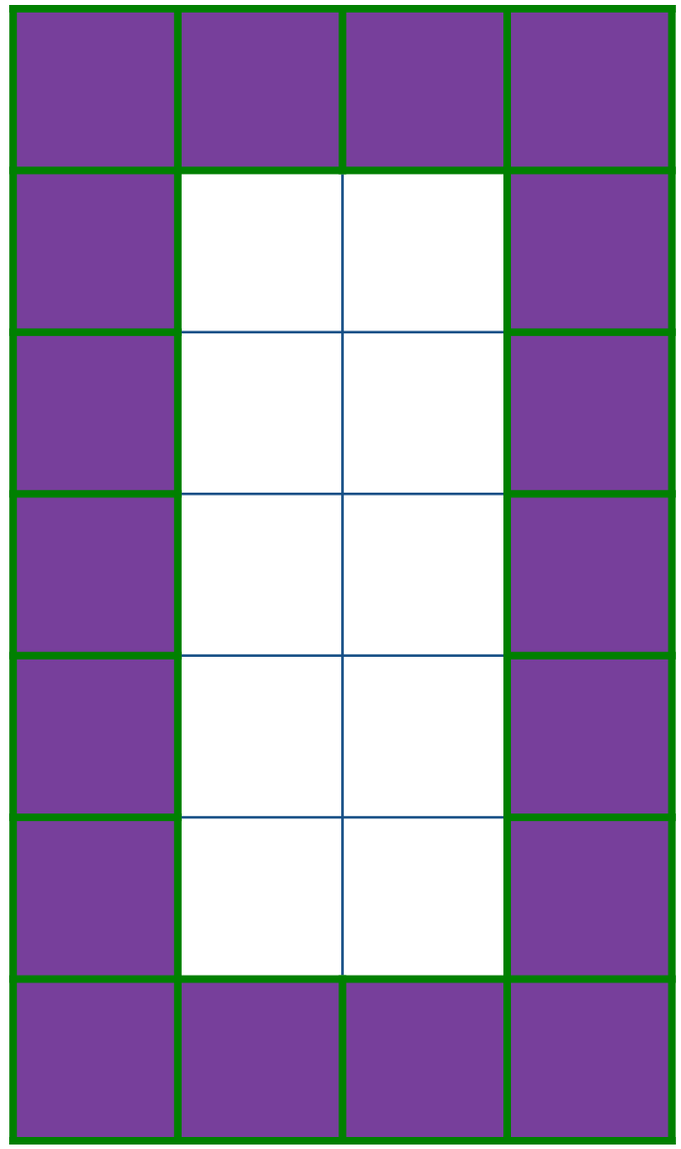
- The recipient chooses when to actually receive the data so it knows that it is in the right state

Tightly coupled parallelism

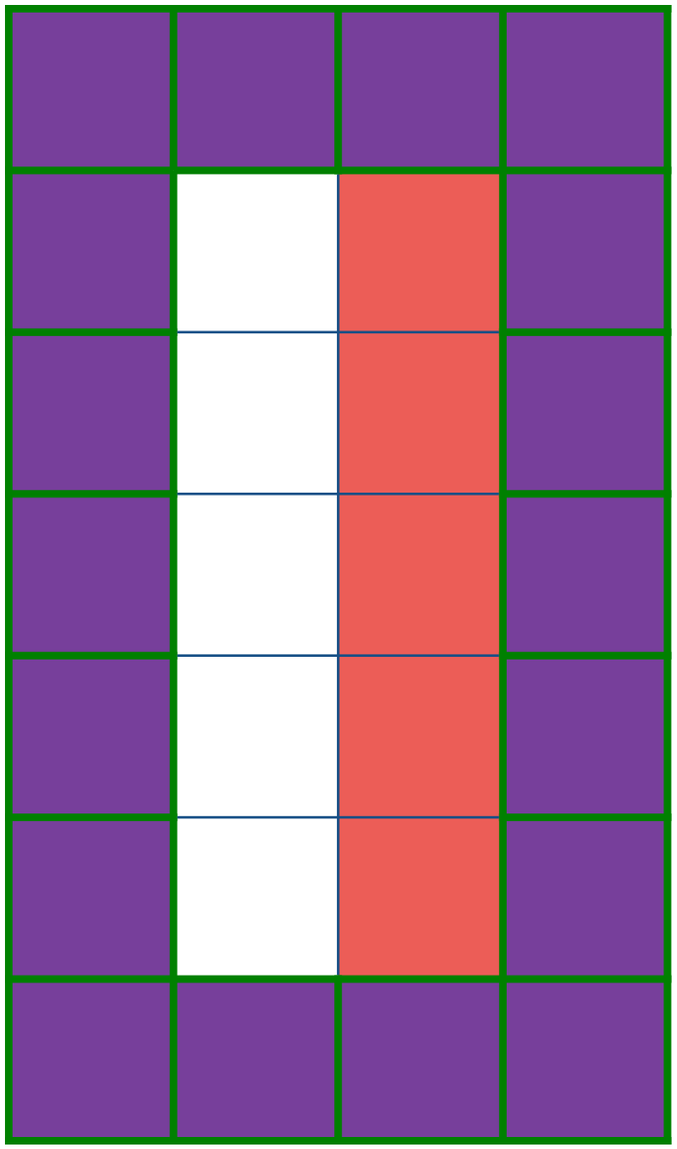


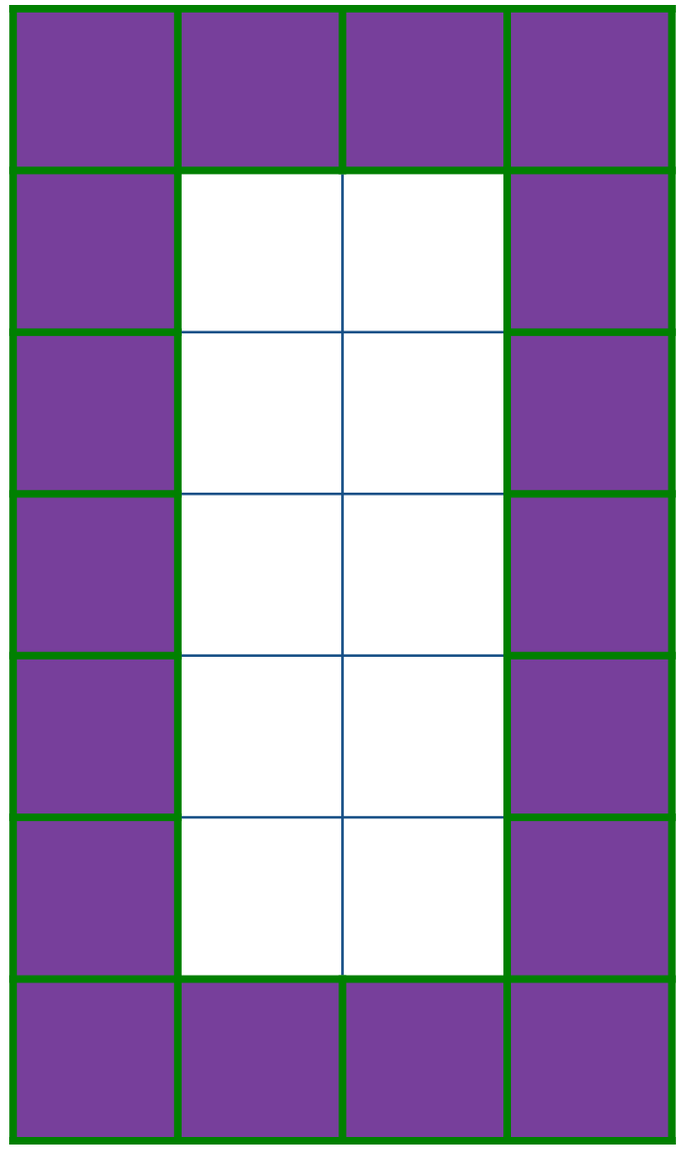
Tightly coupled parallelism



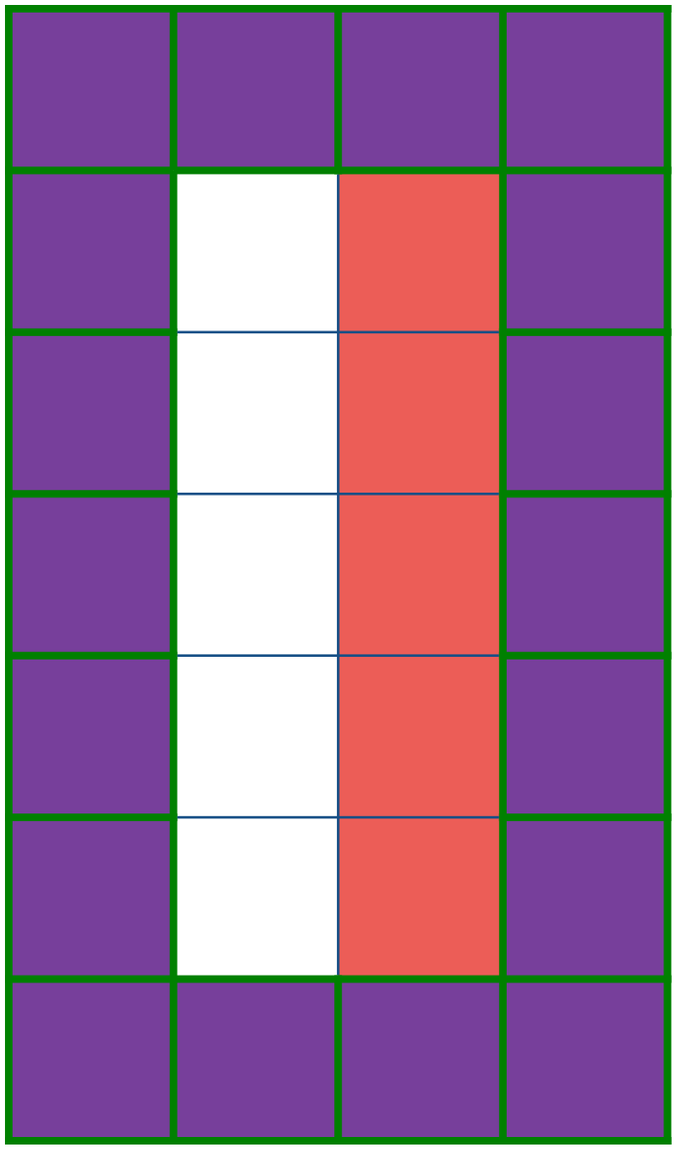


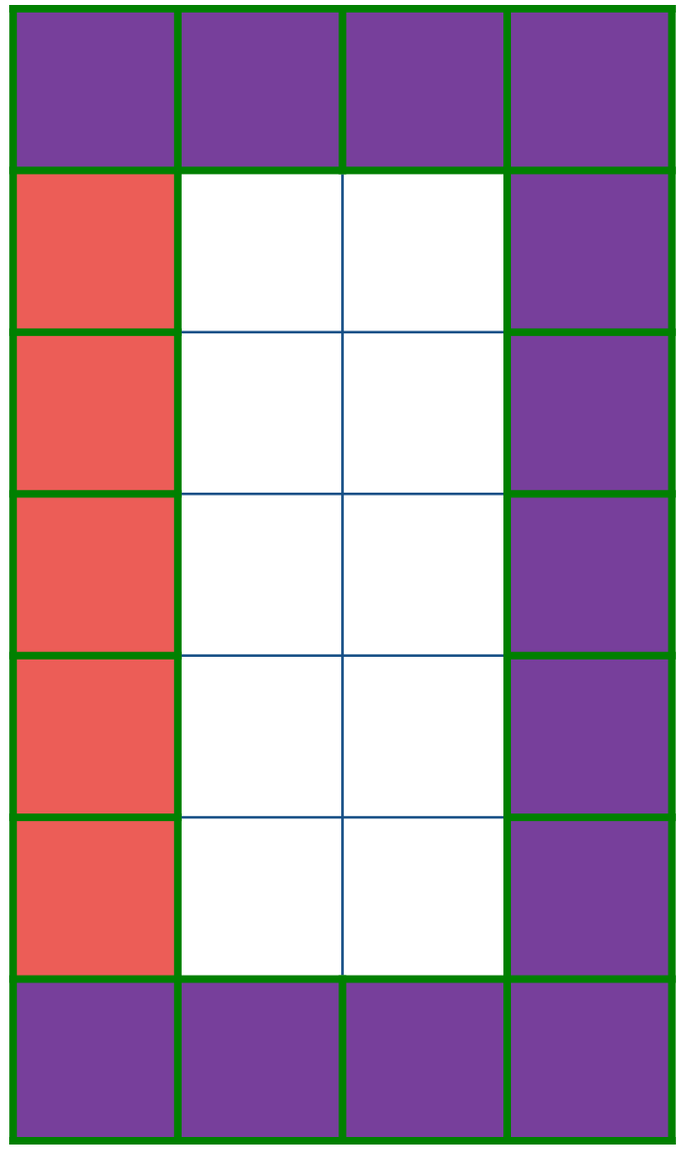
Tightly coupled parallelism





Tightly coupled parallelism





Distributed Memory

```
DO iy = 1, ny
  DO ix = 1, nx
    temp(ix,iy) = 0.25 * (im(ix-1, iy) + im(ix+1, iy) + im(ix,iy-1) + im(ix,iy+1))
  END DO
END DO
!Send temp(1,1:ny) to left processor
!Send temp(nx,1:ny) to right processor
!Send temp(1:nx,1) to bottom processor
!Send temp(1:nx,ny) to top processor

!Receive data from left processor into temp(0,1:ny)
!Receive data from right processor into temp(nx+1,1:ny)
!Receive data from bottom processor into temp(1:nx,0)
!Receive data from top processor into temp(1:nx,ny+1)
im = temp
```

Wrapup

- This section has introduced the three basic models of parallelism
  - Embarrassing parallelism
    - And how to use GNU Parallel to make it easier
  - Worker-controller parallelism
  - Domain decomposition
- The final lectured section of this course covers an introduction to the actual programming methods for the last two