

# Introduction and Analogies

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



# Why analogies?

- Parallel computation feels like something that is very outside of your normal experience
- Actually a lot of the general things that you have to understand make intuitive sense
- Most of the problems, and solutions, to running things on multiple processors are the same as trying to do multiple things in real life

# Limits to analogies

- Be very careful with analogies!
- They can be extremely valuable and quite often a feature that you can identify in the analogy is also a feature in the real problem
- But there are definitely cases where the feature that you identify is an **analogy breakdown** and the feature is not present in the real problem
- If you find these analogies useful then you might want to try to extend them yourselves, but always check to see if the extension apply to the real problem

# Buffet

A	B	C	D
Duck		Chilli	Pudding
Curry	Sushi	Pudding	Pudding
Pause for Drinks	Duck		Pudding
Pudding	Liqueurs		Curry
			Pudding

# Computer Analogy

- Normal operations of the computer
  - Programs are split onto different processors so that they are all used
  - Running two programs at once will normally put one on each processor
- “Embarrassing Parallelism”
  - Solve many problems by running several programs or many copies of one program
  - Don’t want more programs than you have processors or some have to wait their turn, just like at a buffet

# A La Carte

A	B	C	D
Soup			Nachos
Pie	Burger	Curry	Fish
	Ice Cream		Cake
	Coffee	Brandy	Coffee

# Computer Analogy

- Weakly coupled parallel computer programs
  - Mostly things do their own thing with little reference to other things that are doing their work
- You occasionally have to synchronise information between running things to make things happen at the same time
  - Or to make things have a specific sequence

# Prix Fixe

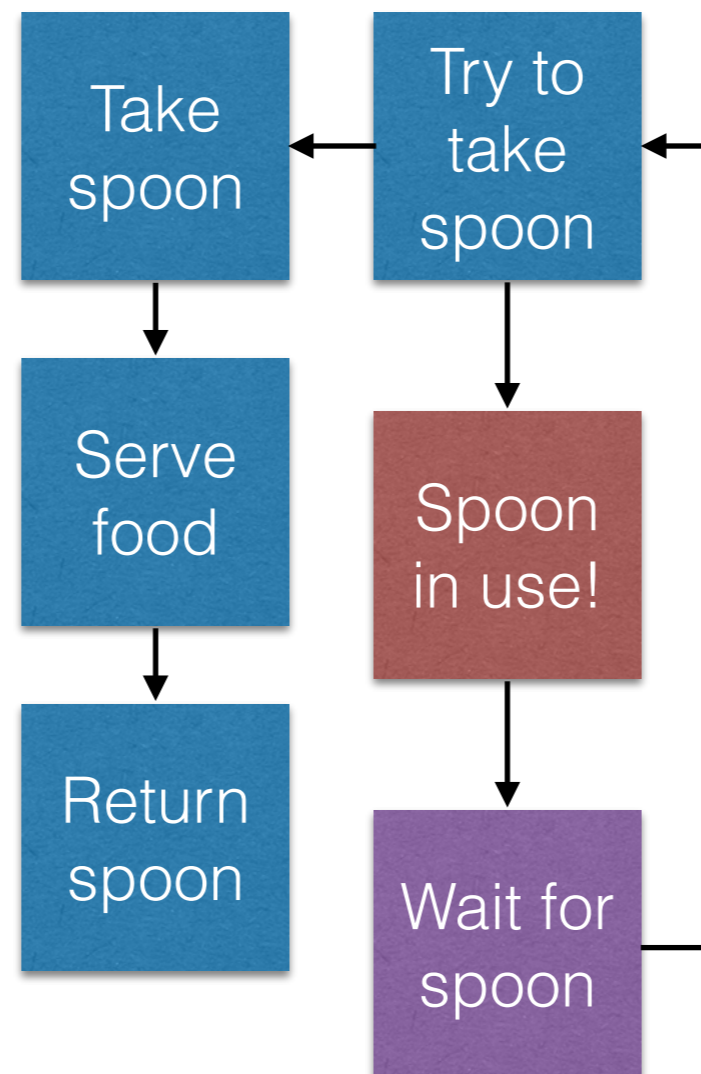
A	B	C	D
Starter	Starter	Starter	Starter
Main Course	Main Course	Main Course	Main Course
Dessert	Dessert	Dessert	Dessert
Coffee	Coffee	Coffee	Coffee



# Computer Analogy

- Tightly coupled parallel processing
- Many sequencing points where things don't advance to the next level until everyone has reached the same point
- Things may be doing different tasks but you have to treat them as if they are all the same because you can't move on until all of them are finished

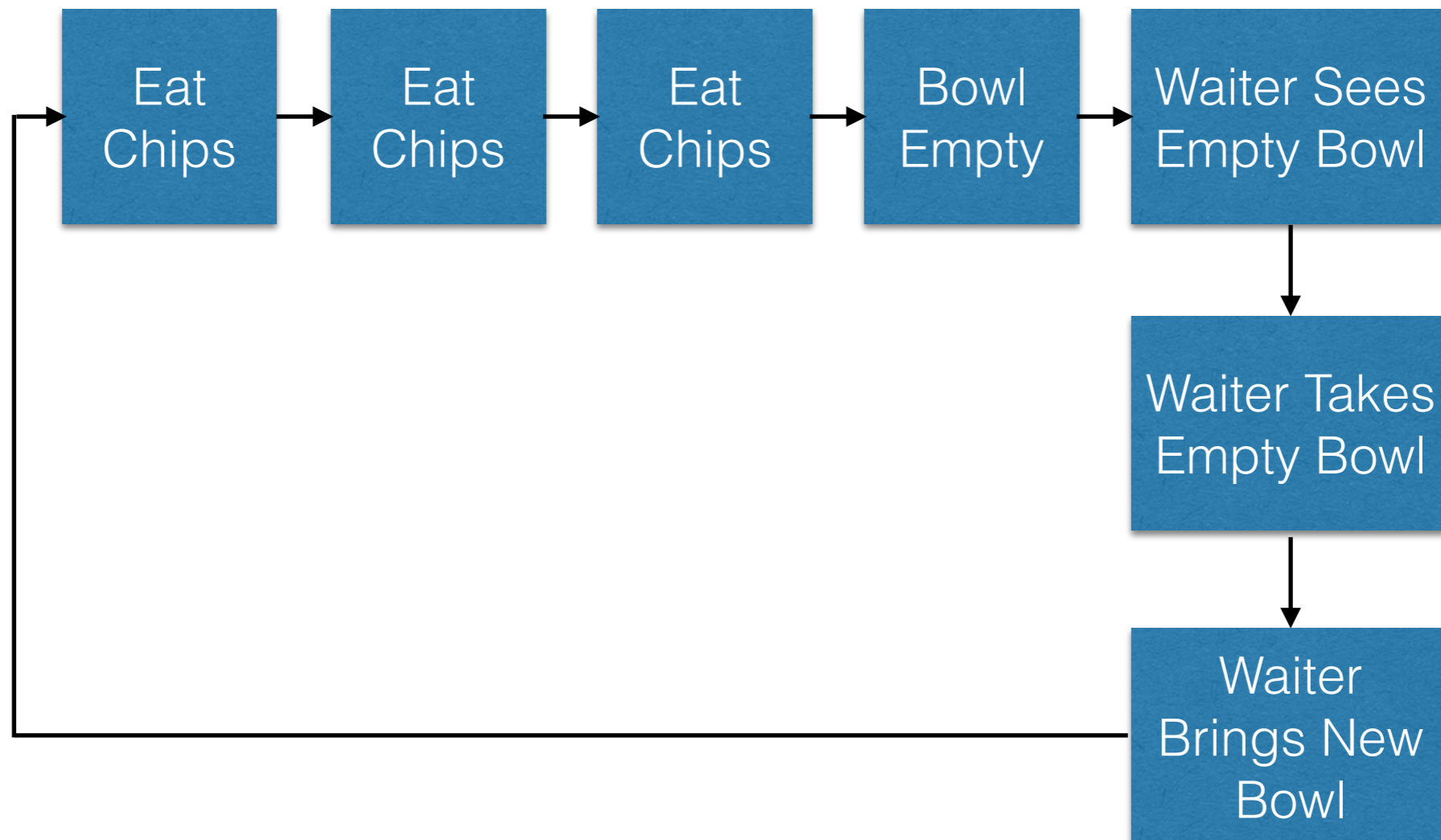
# The Buffet Spoon



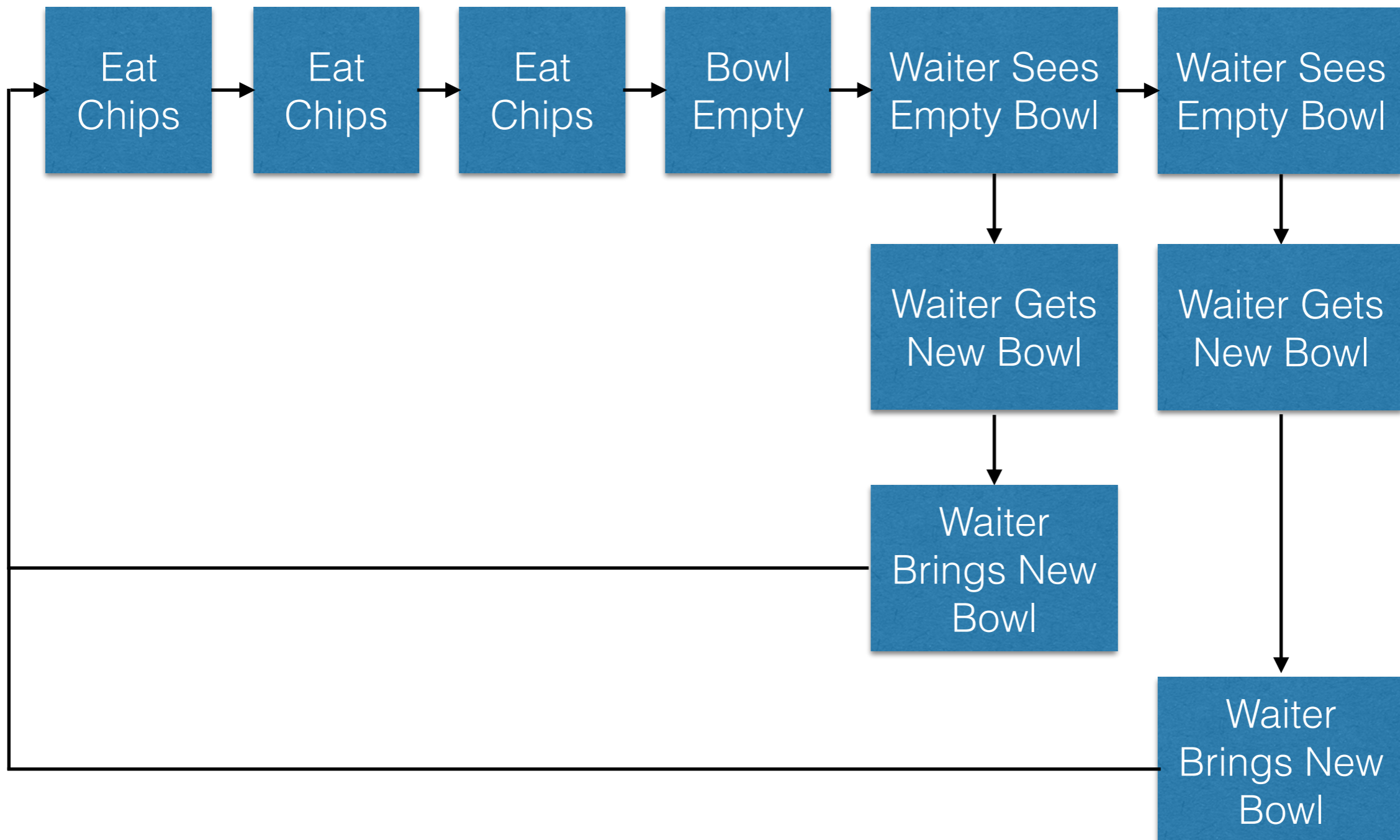
# Computer Analogy

- Lots of things!
  - This is how almost all resources that can only be used by one thing at a time work in computers
- In particular you can deliberately make it happen by using **Mutual Exclusion** objects. Things that only one processor can have at a time. More on this later
- There's actually quite a lot of work devoted to making sure that everything gets its turn with the spoon (actual queues are unusual in computers)

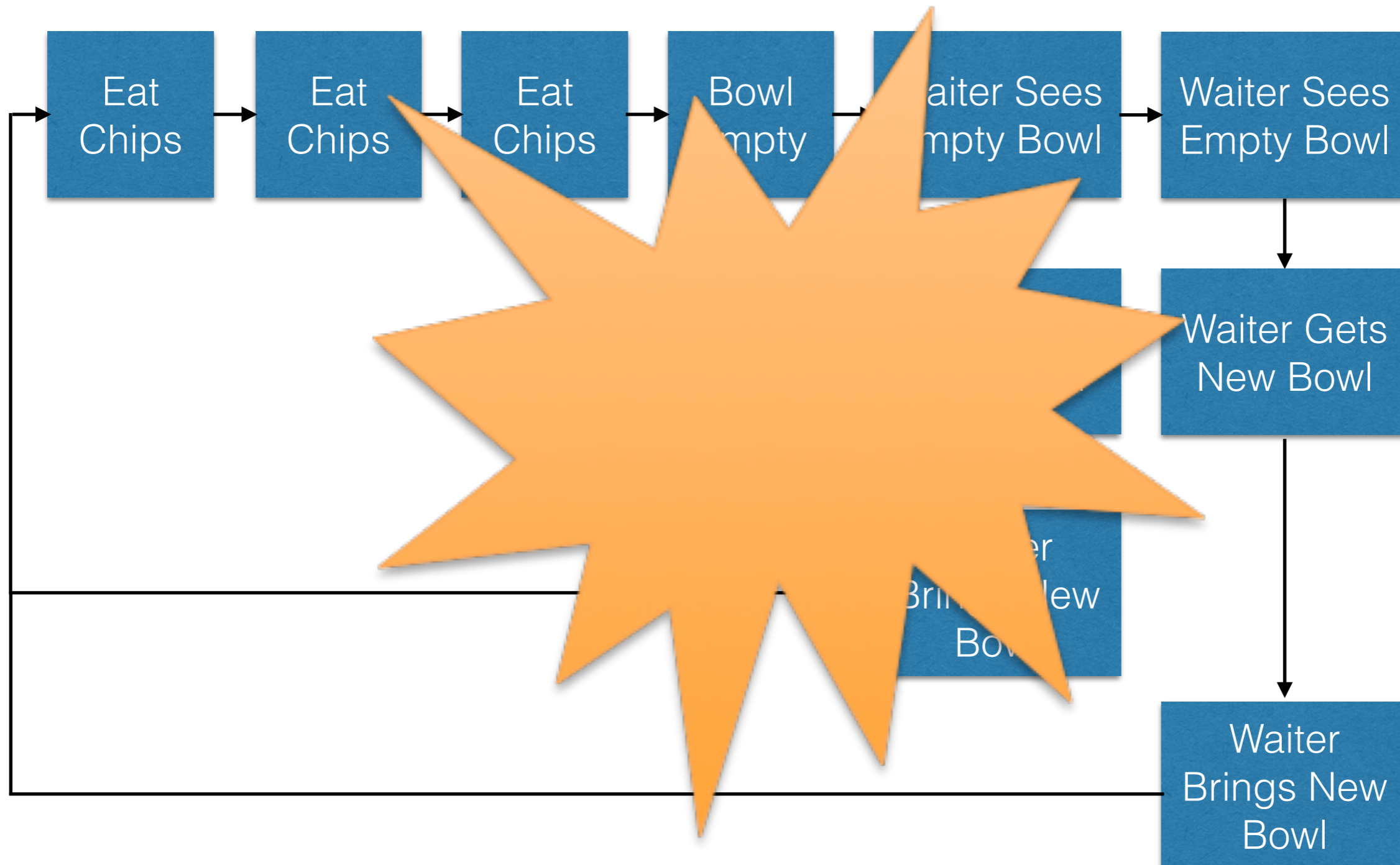
# The Bowl 'o Chips



# The Bowl 'o Chips



# Chipslosion!



# Analogy

- This is an example of something called a race condition
- If a waiter sees you have an empty bowl he will go and get you a new one
- If two waiters both see the empty bowl then you wind up with two bowls of chips
- If the first waiter gets back before the second waiter sees the empty bowl then you only get one bowl
  - Hence race condition. The result depends on how fast your waiter is
- If the first waiter to see you have an empty bowl takes it away then everything will work (no bowl isn't an empty bowl)

# Computer Analogy

- Exactly the same happens with computers
- If two threads are waiting for a signal then things can happen twice if the first one doesn't signal that it has handled it before the second one sees the signal
- How you actually do the signalling involves things called atomic (Greek - Indivisible) operations
  - We'll see these briefly later
- There are other race conditions e.g. two threads setting a variable to a given value - the value will depend on which processor happens to get there last
  - They are always a bad thing



# Computers

The image features a solid dark blue background. The word "Computers" is centered in a white, sans-serif font. At the bottom of the image, there is a white horizontal band with a jagged, sawtooth-like cutout in the center, creating a decorative border.

# Computer Terms

- Probably a lot of this is familiar but it is important that you understand some important terms before we start
- Core - the smallest unit capable of independent general purpose computation. You can get 128 cores in a single chip nowadays
- CPU (or chip) - Central Processing Unit. Physical computer chip that will (nowadays) usually contain multiple cores

# Computer Terms

- Socket - Used to refer to the number of physical chips in a computer ("a 2 socket workstation")
- Node - A single complete computer but usually one that is part of a **cluster**
- Cluster - A collection of computers connected together for parallel processing

# Computer Terms

- RAM - Random Access Memory. The part of the computer that stores the data that you are working with
- Hard Drive/Disk - Long term stable storage medium that persists data after the computer is switched off.

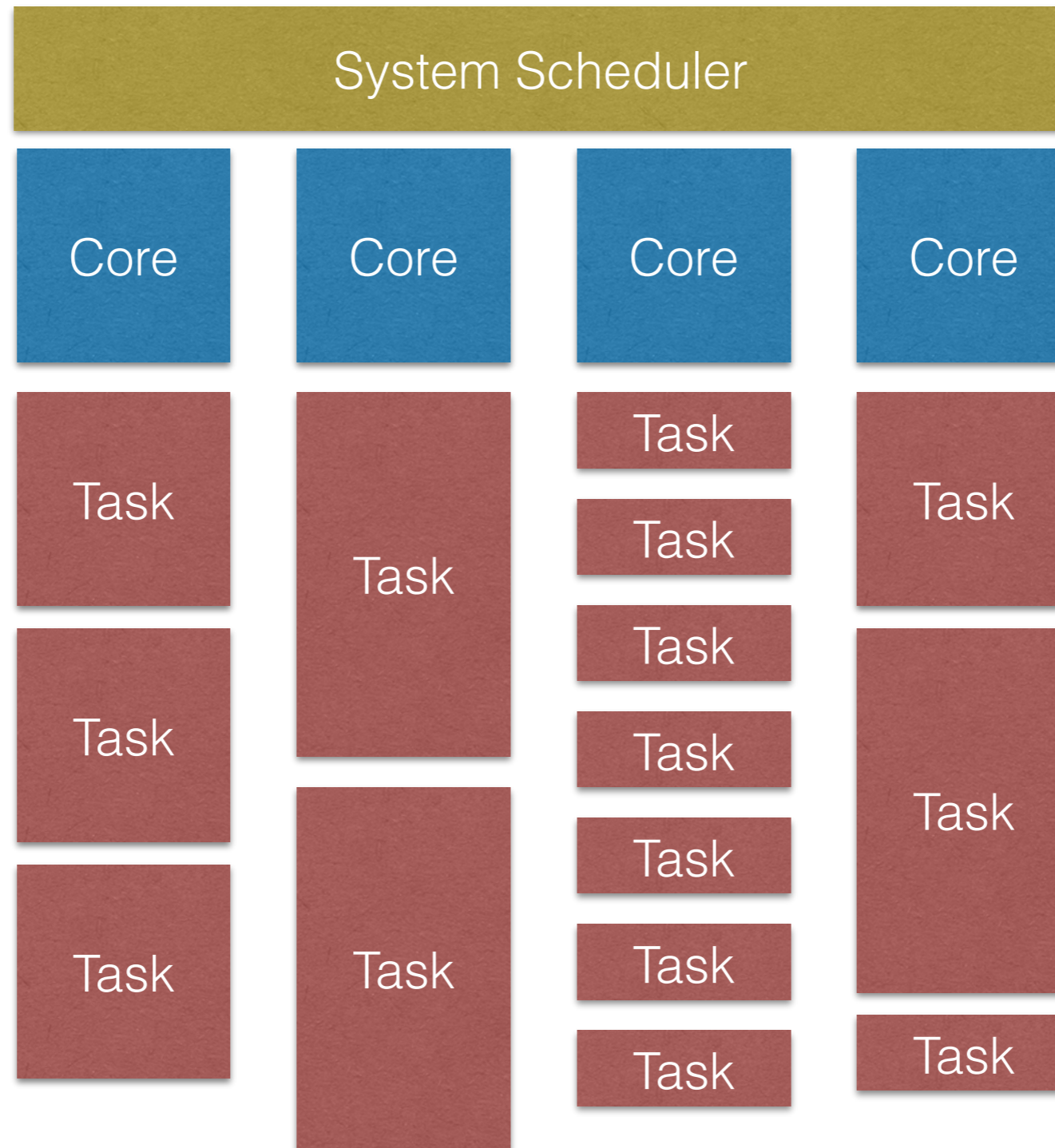
# Computer Terms

- Process - Technically the name for a running program. Processes do not interact directly with each other (normally, there are mechanisms to do it) and will generally run on separate cores to each other if possible
- Thread - A part of a process created to run independently. Threads are connected to a single process and a given process can have many threads.

# CPUs

- Mostly we're going to be talking about cores here because they are what actually does the parallel processing
  - The idea is to move from using one core to many cores
  - Sometimes very many
- CPU behaviour in a computer doing many things is very like that buffet example

# 4 Core CPU



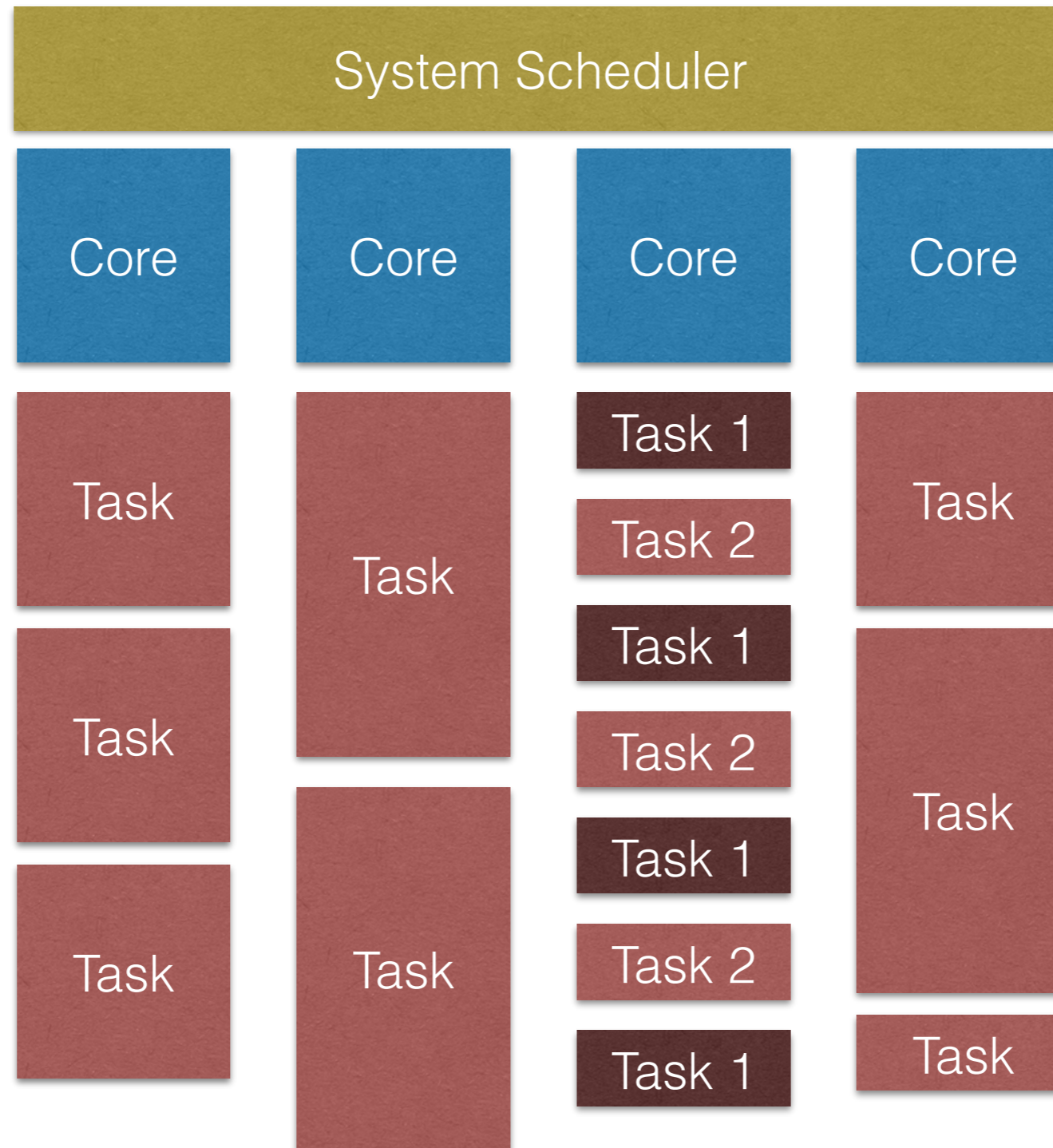
A possible, heavily simplified schedule

# CPUs

- We mentioned that in the buffet people have to wait if they both want to get the same food
- Computers don't quite work like this (analogy failure)
  - If they did then as soon as you had one program running per processor in your computer you would have to wait for one of them to finish before you could do anything else
- Timeslicing - suspend one program to allow another to run



# CPU Multitasking



A possible, heavily simplified schedule with one more task than available CPUs

# CPUs

- Time slicing is brilliant for problems like web servers.
  - The thread that is serving a website to any one person is mostly not doing much (waiting for the person to load new data)
  - So having multiple threads being interleaved works quite well even when you have more threads than you do cores
- Works OK for interactive use as well because humans are much slower than computers
- This isn't generally true for academic codes where all threads are working flat out at all times

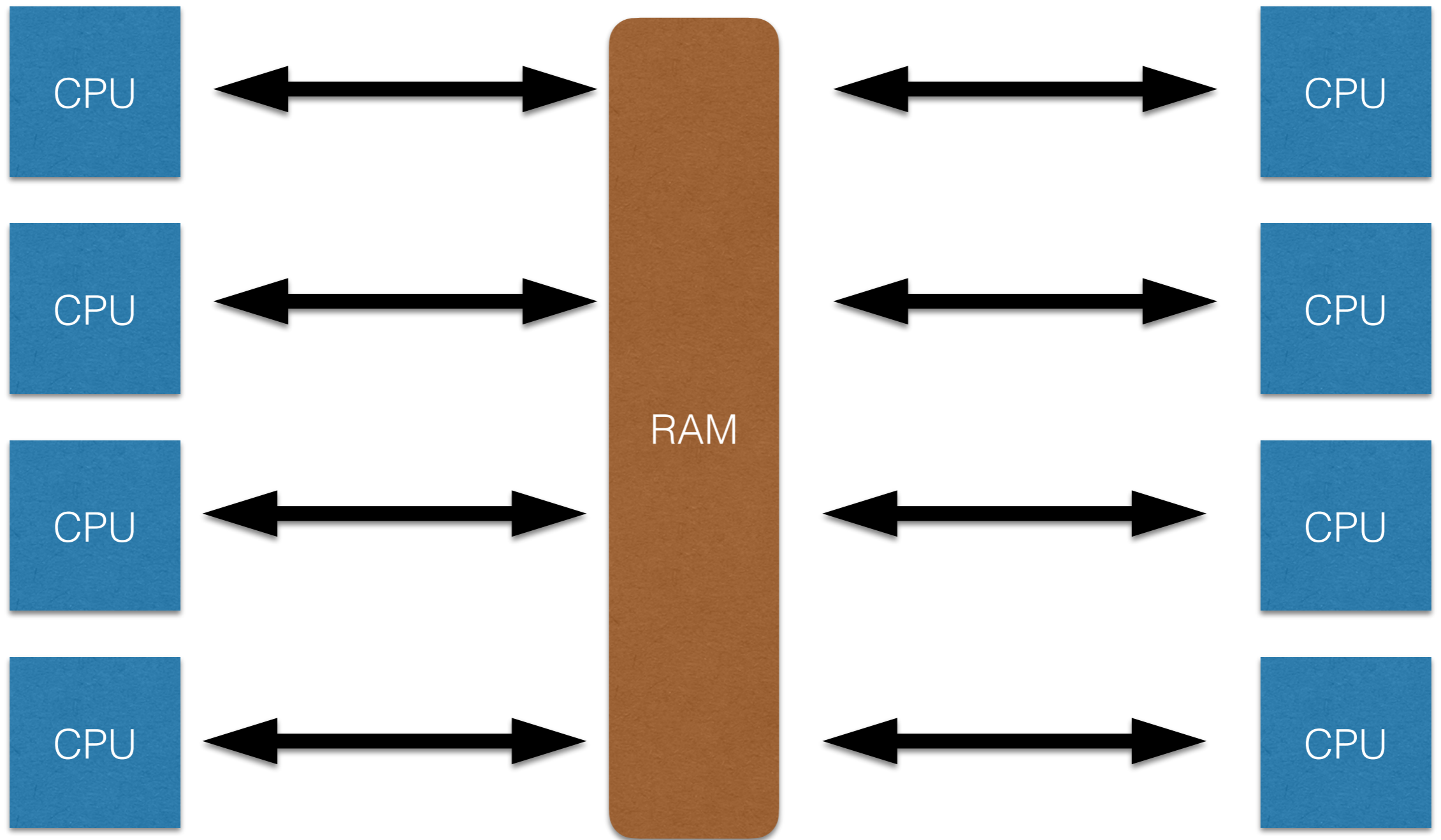
# Memory

- The other really core element of a computer is memory
- Memory has a speed, and faster is better (broadly) but mostly you take what you are given in the parallel programming world
- From a parallelism perspective there are two classes
  - Shared memory
  - Distributed memory

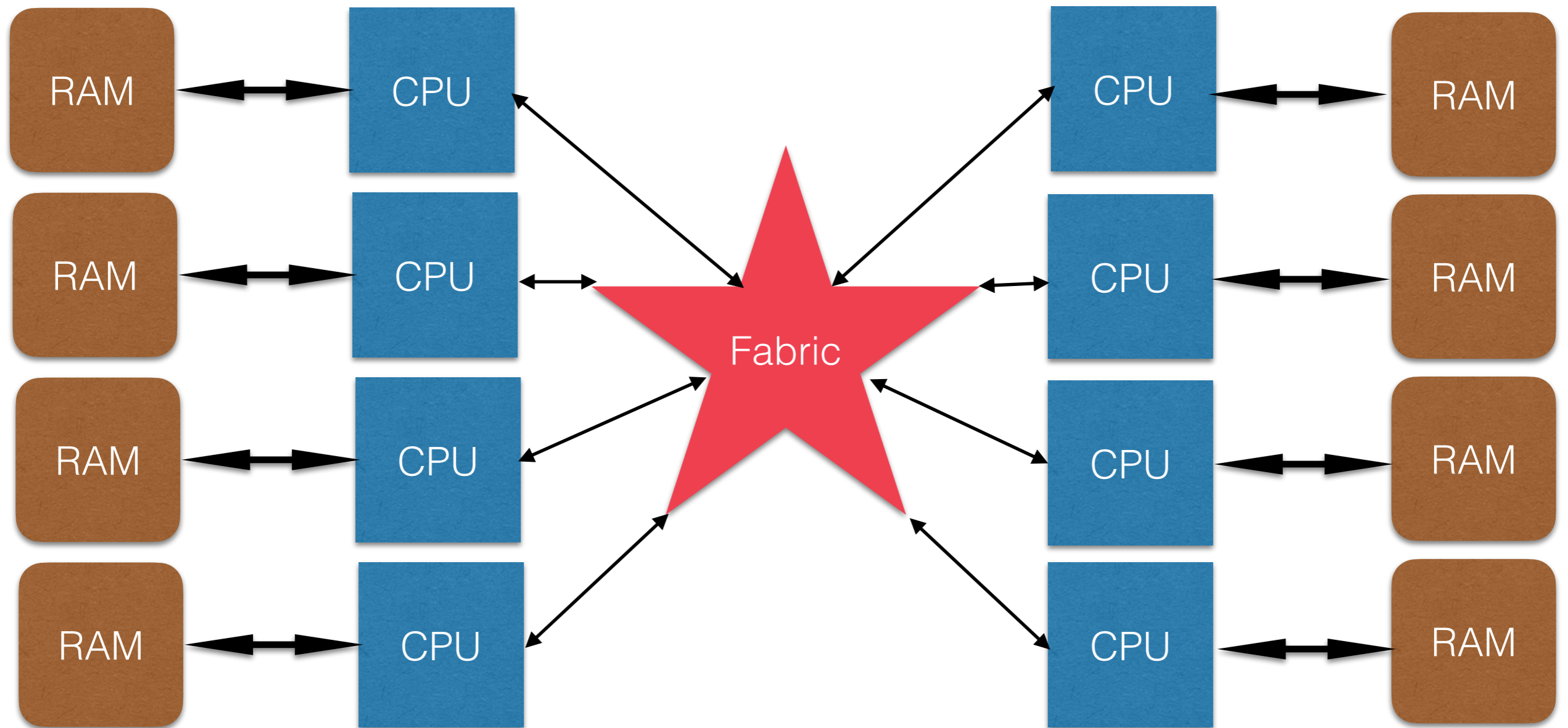
# Memory

- Shared memory
  - Multiple processors connected to a single memory system
  - Easy for different processors to work on a single problem, but have to make sure that you avoid things like race conditions
- Distributed memory
  - Each processor has its own memory and has to explicitly communicate with other processors to work with them
  - Harder to program but can work across multiple computers, so this is what the largest HPC codes are programmed for

# Shared Memory




# Distributed Memory



# GPU

- GPUs are often used for high speed computation nowadays
- They are parallel processors, usually capable of running thousands of computations in parallel
- We're not going to talk about them here
  - They are different to CPUs and are mostly suitable for problems that can be split up thousands of ways

# Unconnected task parallelism





# Unconnected task parallelism

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

# Unconnected 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
  - Also can run into problems if you are doing enough work with files - hard drives are slow compared to computers

# GNU Parallel

- Very good official tutorial at [https://www.gnu.org/software/parallel/parallel\\_tutorial.html](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
```

# 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

# Connected task parallelism



# 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
- One way of doing it is **Futures** or **Promises**



# Futures

- 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
- When you ask a future to give you the result of its calculation it will wait until the result is available
- So if you make one future’s inputs depend on the results of another future you can say to start both and the second one will only start when the first has finished
- You can also manually check the results and conditionally run other calculations if the result is “interesting”

# Futures

- Futures are built into Python and C++ (although in C++ you combine them with things called **async** objects)
  - They only run on a single machine
- The **Dask** library for Python has distributed futures that can run on clusters
- There are libraries for the same idea in other languages, but not one preeminent one

# Tightly Coupled Parallelism



# Tightly Coupled

- The most “extreme” form of parallelism is tightly coupled parallelism
  - Taking one problem and splitting it up into parts so that more than one processor can work on it
- Approaches based on memory type
  - Shared memory – processors work on different bits of the shared problem in memory
  - Distributed memory - processors communicate by exchanging information explicitly between computers

# Scaling

- Both of the previous forms of parallelism are task based
  - You are running a single problem on a single processor even if you have to wait for one problem to finish before starting another one
- With tightly coupled parallelism no part of the problem can proceed until the bits that it depends on are at the same point
- This introduces the idea of **scaling** - if I double the number of processors how much does the runtime drop
  - $2x$  processors =  $1/2$  runtime is perfect scaling

# Libraries and Packages

- Quite often there is a “best” solution to how to solve a given problem in parallel
- Libraries have been written to solve these in parallel already
- Look for a library to solve your existing problem
- If your problem can be written in vector/matrix form then **lots** of libraries and lots of packages have been written to solve your problem
- Use these before writing your own code!

OpenMP

A decorative graphic at the bottom of the slide, consisting of a solid blue horizontal bar that transitions into a white background with a blue zigzag pattern.

# OpenMP

- OpenMP allows for almost completely general parallel programming
- But by **far** the most common use of OpenMP is to split loops up so that different bits of the loop are handled by different processors
  - There's an obvious limitation to this: only loops that have independent iterations can be parallelised over
  - If you have a loop to advance a quantity in time then you **can't** parallelise that since iteration 2 depends on iteration 1, which depends on iteration 0 etc. etc.



# OpenMP

- On this level OpenMP works by adding directives to your code telling it to split a given loop up or not
- Easy to retrofit into codes that do most of their work in loops

MPI

# MPI

- MPI is the Message Passing Interface
- You explicitly write code for both sides of a message exchange
  - Source says "I have this data to send"
  - Destination says "I want to receive this data"
- System ties up source and destination and actually sends the data

# MPI

- You have to design your algorithm and your code to explicitly use MPI
- It is not easy to retrofit MPI to an existing code, but it isn't that hard to program
- Allows your code to run on distributed memory systems
- Can't guarantee how many processors it will scale to, but MPI will let you scale as far as your algorithm will