SOFT354 - Revision notes

Date: 03-12-16

Static allocation

- Size of array must be known at compile time
- Space is reserved in the program memory map

Allocation in host memory: `float h_array[10];`

Allocation in device memory: `__device__ float d_array[10];`

Dynamic allocation

- Size of array can be calculated at run time

Allocation in host memory: `float* h_array = (float*)malloc(10*sizeof(float));`

Allocation in device memory: `float* d_array = (float*)cudaMalloc(&d_array, 10*sizeof(float));`

- If you use `malloc` inside the kernel it will allocate memory on the **device**
- If you use `cudaMalloc` inside the kernel
  - Every thread would allocate enough space to hold one copy of the array in device memory
  - And each thread accesses a different bit of it

Copying to static device arrays

- When you use **static allocation** for device memory
  - `__device__ float d_array[10];`
- The variable `d_array` is not a pointer to a memory address on the GPU
  - **it is a symbol**
- Therefore you cannot use `cudaMemcpy`
- Have to use `cudaMemcpy*Symbol`

Accessing static memory from a kernel

- If device memory was allocated statically in the host
- It allows it to be accessed globally on the device
- It doesn't require time consuming calls to memory allocation functions
- Automatically frees itself in memory after use

Pinned Memory

- Technique for speeding up RAM < - > GPU memory transfers
- Transfers between RAM and GPU use **Direct Memory Access** to do the copying without working the CPU
- Where by a section in RAM is 'pinned' for the GPU as RAM uses paging to swap around memory
- This means that the pages aren't swapped so it allows for faster transactions as it doesn't require finding the memory address
- Because of paged virtual memory
- The address you get from `malloc()` won't correspond to a physical address in RAM but instead a `page` that can be moved around
- So it copies it first into DMA where it won't get moved
  - Then that allows for quicker transfer
- Transfers can happen entirely with this
  - big speed increase

`cudaMallocHost`

- like `malloc` it dynamically allocates an array in RAM
- unlike `malloc`, the memory will be **pinned**
  - Memory calls to `cudaMemcpy` will be faster

Uniform cache

- The uniform cache is an on-chip (part of each SM) cache that is designed for **broadcasting data**
- If multiple threads access the same address in the uniform cache at the same time
  - The data is sent to them all simultaneously
- Threads can't change the value of data in the uniform cache
- Similar to L1 cache
- SM has its own L1 and uniform cache
- It is used for broadcasting out data

Registers

- Fastest, accessible by a single thread
- The fastest
- Local variable inside kernel go into registers where possible
- Two exceptions:
  1. If the index accessing the array using a variable then it is stored in local memory
  2. If you use more than the available register space

L1 Cache / Shared memory

- located in an SM
- Shared between threads in that SM
- Extremely fast
- Individual to each SM
  - L2 is shared between all SMs
- **shared:** you have to program its use
  - Allocated by using `__shared__` modifier
- Can control how it is used
- Every block of thread has this area to communicate with threads that are in the same block
  - Each block has a separate copy of shared variables
  - All threads in a block can access their copy of variables but not other SMs
- Common pattern:
  - All threads in a block need the same chunk of global memory
  - Each thread loads a list of the global data into shared memory in parallel

L2 Cache

- Shared between all SMs
- Much faster than RAM but slower than L1
- Bigger than L1

Global memory - RAM

- Biggest and slowest
- Shared between all SMs
- **Only memory we can read and write from host**
- Only memory that you can directly access from inside the host - using `cudaMemcpy`
- Slowest type of transfer
- Can speed it up using DMA which pins memory
- Allocate global memory
- Global memory has smaller regions for specific uses:
  - **Constant memory**: If we put variables here they will be cached for broadcast in the SMs' uniform caches
  - **Local memory**: Any of a thread's local variables that can't go in registers go here

Threads, Blocks and warps overview

- **Threads** are organised into **blocks**
- **Blocks** are organised into **Grids**
- The **thread** in a **block** and/or the **blocks** in a **grid** can be organised in 1D, 2D or 3D structures
- Whole **blocks** are allocated to SMs as they become available
- When a **block** is allocated to a SM, it is divided into **warps** of 32 **threads**
- We want to avoid the situation where all **warps** in a SM are waiting and not runnable
- Two strategies:
  1. Maximise the number of **warps** in an SM (*its occupancy*) by optimising block size
  2. Reduce the time spent waiting for memory access by using *coalescing* and *shared memory*
- A **block** consists of multiple **threads**
- A **grid** consists of multiple **blocks**

Optimisation techniques:

- Increased thread **occupancy**
- **Coalecsed** memory access
- **Shared** memory

2D blocks and grids

- If you use 32x32 blocks of 1024 threads:
- How many blocks would you need for a 2D array of 2,450 x 3570?
- x direction: 2,450 / 32 = 76.5 = 77
- y direction: 3,570 / 32 = 111.5 = 112
- **Cannot have half blocks so round up**
- If you round up, you need an if statement to catch the threads that will be out of bounds
- 2D and 3D blocks and grids will all converted to 1D by CUDA automatically

Block assignment

- Each SM has one or more **blocks** assigned to it ('resident') at any given time
- Only whole **blocks** can be assigned to SM

    - Therefore all threads in a block will run on the same SM
- The number of blocks/threads that can be resident in an SM depends on the devices Compute Capability

Warps

- When a SM is assigned a block of threads it breaks it down into warps
- Each **warp** is a set of 32 threads that will be executed in parallel
    - Threads are assigned to warps by dividing the linearised blocks into 32 thread chunks

Warp Scheduling

- Understanding warp scheduling means you can improve performance
- At any given time, a warp can be in 3 states:
    - **Running:** Currently being executed by the SM
    - **Waiting:** Can't run because it is waiting on something, usually memory access
    - **Ready:** Not running but not waiting on something
- Ideal situations is when one is running, there are two more ready and more waiting
- Worst situation is to have all SMs waiting, effectively meaning SM is idle

Reducing stalls

- Most effective way to reduce stalls (assuming that it is waiting on memory)
- Is to use memory more efficiently
- Coalesced memory reads and shared memory
- But also important to maximise the number of warps in an SM
    - **its occupancy**
- The more warps there more chance there's one ready to be executed

Calculating optimal occupancy

Need:

- Max no of threads per block
- Max no of resident blocks per multiprocessor
- Max no of resident threads per multiprocessor

Following block sizes:

- 8x8
- 64 threads per block (2 warps)

- Each SM can only contain 8 blocks
- Max number of resident threads is `8 * 64 = 512`
  - Much less than the capacity of 1536
- 16x16
- 256 threads per block (8 warps)
- Each SM can only contain 8 blocks
- Max number of resident threads is `6 * 256 = 1536`
  - Perfect fit
- 32x32
- 1024 threads per block (32 warps)
- Max cap is 1536
- So cant have any more than one

Coalesced memory access

- If threads in a warp simultaneously access memory addresses that are close together (128 bytes)
- Then the accesses are coalesced into one transaction
  - Much faster
- You have to plan how data is stored in memory so that the data that the threads need are next to each other

Vector major order

- Each full vector is placed next to another from start to end element
- To add all first elements of the vectors to a scalar
- threads have to skip the length of the vector to get to the next first element
- Means wasted N number of loaded elements
- To access `globalId * N + i`
- Global Id = thread
- N = size of vector
- i counter for each element in array

Component major order

- Each first element is stored next to each other
- Then each second element, each third, etc.
- To add all elements by a scalar value
- Can access all consecutive memory addresses for the number of vectors to add
- To access: `i * M + gloablId`

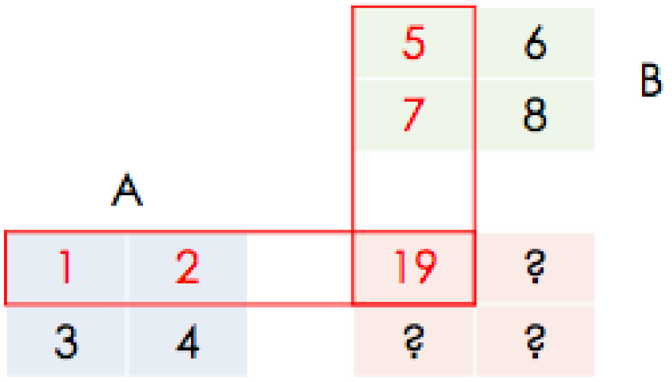
Divergence

- Try to avoid branching statements in kernel code
- Only occurs when threads in the same **warp**
- All threads in the same warp share the same instruction pointer
- Therefore, all threads have to be on the same instruction
  - Branching causes threads to take different paths through the code depending on their value
- This can make it very inefficient
- The code should run serially at all times possible
- Try to change so only certain blocks access code meaning that all other warps in other blocks don't have to
- If threads do have to wait for other threads after divergence
  - They're marked as de-active
  - until they are all at the same point

`Syncthreads`

- This causes threads to pause and wait until this point
- Waits until all threads are together
- Resumes code
- Can be dangerous
  - Don't write it in a branch as some threads may never reach it

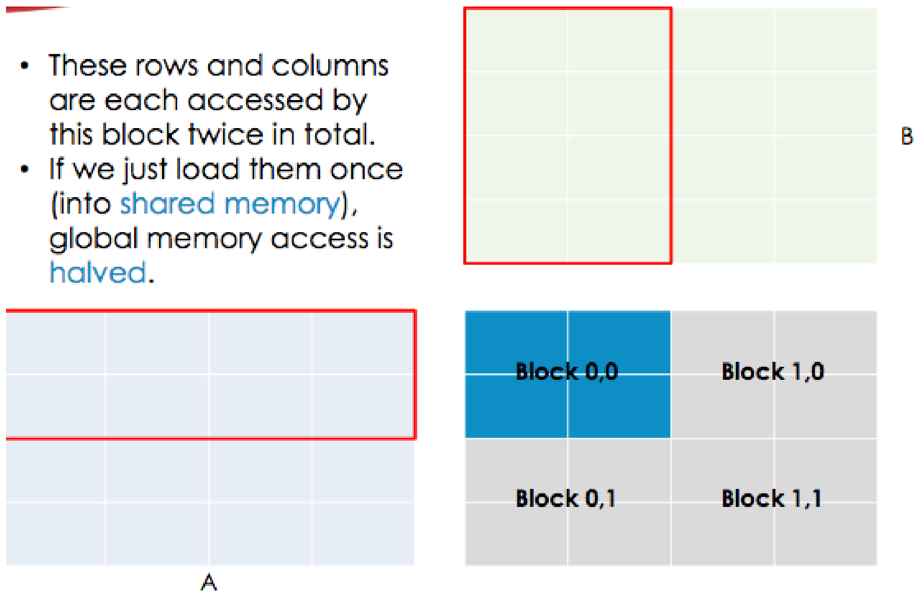
Matrix multiplication



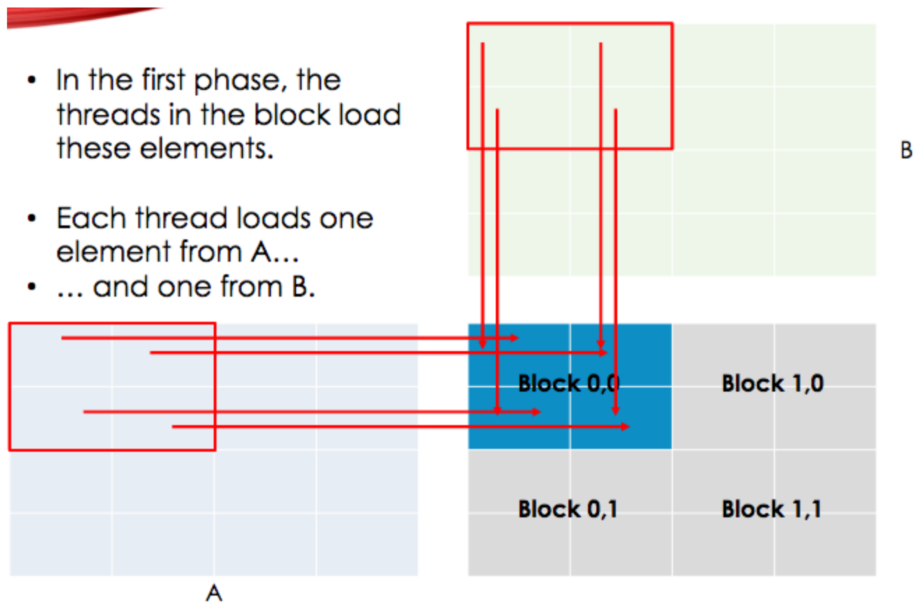
- for first element in top row, take first row and first column and get the dot product - for second element in top row, take first row and second column - Repeat for second row

Shared memory matrix multiplication

- Each row of the block
- The threads need to access the entire row of the matrix that they're multiplying
- Likewise with the columns
- Therefore, a whole block's threads need to access both columns and rows twice
- If you load first two rows and first two columns for first



- These rows and columns are each accessed by this block twice in total.
- If we just load them once (into shared memory), global memory access is halved.



- In the first phase, the threads in the block load these elements.
- Each thread loads one element from A…
- … and one from B.

- In the first phase, the threads load the size of their block that they're calculating from each one of the matrix
- It calculates half its dot product
- If the matrices were 6x6 then n = 6 and the blocks were 2x2
- we would need 3 phases
  - Phases are the turns in which the thread loads its elements into the block from the memory
- num of phases = n/m

Compute to global memory access (CGMA)

$$CGMA = \frac{\#\, floating\ point\ operations}{\#\, global\ memory\ accesses}$$

- How many floating point operations will a block do?
- How many global memory accesses will it do

E.g: for a convolution mase

- Db = 4 = No. of threads in a block
- M = 5 = Mask size

|2|4|**5|2|3|7|2**|9|1

- *In bold is the no of threads in the block = Db*

How many computations?

- Each thread does M multiplications
  - one for each of the mask
- And then does 4 additions = M - 1
- There are Db threads in a block
  - So:

```
Db * (M(M-1)) = Db * 2M - 1
```

How many global memory accesses?

- Each thread loads M elements
- Also has to store 1 element (the result)
- Db threads in a block
  - So:

```
Db * (M + 1)
```

$$\mathrm{CGMA} = \frac{D_B(2M-1)}{D_B(M+1)} = \frac{2M-1}{M+1} \approx 2$$

Rubbish!

Using shared memory:

- Only the global access memory part of the ratio changes
- Each block as a whole loads Db + M - 1
- Has to store Db (its results)
  - So:

```
2Db + M - 1
```

$$\mathrm{CGMA} = \frac{D_B(2M-1)}{2D_B+M-1} = \frac{2M-1}{2+\frac{M}{D_B}-\frac{1}{D_B}} \approx \frac{2M}{\frac{M}{D_B}} = \frac{2MD_B}{M} = 2D_B$$

Task parallelism

- A task is split up into different sections where different workers work on different parts of the same task at the same time

Data parallelism

- A task is processed from start to finish by one thread/worker
- But potentially different data is used for each task

Single instruction stream, single data stream (SISD)

- The CPU does **one operation** at a time on **one piece of data**

Single instruction stream, multiple data stream (SIMD)

- **Multiple cores** that process **multiple data streams** in parallel
- **BUT** each core executes the same instruction at once

Multiple instruction stream, single data stream (MIMD)

- The **same data** is read from memory into **multiple processing units** each of which have their own instructions

Multiple instruction stream, multiple data stream (MIMD)

- Each processing unit can **access different data from memory** and run **different instructions**

Shared memory system

- Each processing core can **directly access** the **same memory**

Distributed memory system

- Each processing core has its **own private memory**
- It can only access **another's indectly**

Hybrid memory system

- **Groups** of processing cores **share memory**
- **But** memory is **not** shared between groups

Speed-up

- How long does a parallel version of a program take to run vs a serial version

- T_S - how long the serial program takes to run.
- T_P - how long the parallel program takes to run.

$$S = \frac{T_S}{T_P}$$

Linear speed-up

- If parts of the program can be done completely independently
- Adding more processors means that the task becomes faster
- Good for scaling as you can just add more processors

$$T_P = \frac{T_S}{p} \quad \text{so} \quad Speedup = S = \frac{T_S}{\frac{T_S}{p}} = p$$

Parallel efficiency

- As the processor count increases
  - So do the associated overheads
- Although speed up increases
  - It gets less so with each processor added
- Measure it with efficiency
- For linear speed-up, `E = 1`
- For sublinear speed-up, `E < 1`

$$E = \frac{Speedup}{\#\,Processors} = \frac{S}{p}$$

Amdahl's Law

'There is a limit to how fast you can do something even if you add more processors'

- It is limited by the time spend processing the section of code that can't be serialised
- For the same problem size, as the number of processors increases speed-up is limited to:
    - *where r is the proportion of code that cannot be parallised*

$$S = \frac{1}{r}$$

- No matter how many processors we have the code will always be limited to:

1 / proportion of code that can't be parallised

- It assumes you want to process the same amount of data as quickly as possible by adding processors
    - *task-parallel approach*

Gustafson's Law

'By adding more processors you can always process more data in a given period of time'

- Assume that the problem size (amount of data to be processed) increases with the number of processors (p)
- Divide parallel program execution time into two parts:

$$T_P = \underbrace{\boxed{a}}_{\text{Time taken in serial execution}} + \underbrace{\boxed{b}}_{\text{Time taken in parallel execution}}$$

Example:

- CUDA program that runs in parallel across 1,00 cores, processing 1,000 pieces of data
- Starting up CUDA, copying data to/from GPU, launching kernel takes 100ms
  - `a = 100ms`
- The kernel runs (on all cores in parallel) for 500ms
  - `b = 500ms`

$$T_P = 100 + 500 = 600ms$$

- What is Ts? - Whatever the cores are doing to the data takes 500ms - So if we only have one core, each piece of data has to be processed sequentially:

$$T_S = 100 + 1000 \times 500 = 500100ms$$

- **If the problem size scales with the processor count (p)**
- The speed-up is limited to:
  - *where r is the proportion of the program that can't be parallelised*

$$S = p + r - rp$$

Message Passing Interface MPI

- MPI is an application programming interface (API) for *distributed memory* parallel programming
- Usually MPI follows a *single program multiple data (SPMD)* approach
  - Write one program and spawn multiple copies of it:

```
mpiexec -n 100 program.exe
```

- But can also be used for a *multiple program multiple data (MPMD)* approach:

```
mpiexec -n 50 program.exe : -n 50 otherProgram.exe
```

MPI terminology

- **Communicator**: A group of processes that can talk to each other
  - Default: `MPI_COMM_WORLD` - contains all processes
- **Rank**: An integer uniquely identifying a process with its communicator
- **Tag**: A (user-defined) integer attached to a message that can be usd to indicate the type of message

Threads v Processes

- Every individual program is a separate *process*
- Processes cannot normally access each others' memory
  - *distributed memory*
- Communicate in various ways
- Each process has one main *thread* of execution
  - But has the ability to spawn additional ones
- Each thread can do something different
  - *run different bits of code*
- All threads can access their processes' memory
  - *shared memory*

Threads

- 'lightweight'
- Low memory overhead
- Can switch between them quickly
- Shared memory means data can be exchanged very easily and quickly
- All need to be on the same machine

Processes

- ‘heavyweight’
- Higher memory overhead
- Switching takes longer
- Distributed memory communication is harder and slower
- Can be running on different machines
- Communicate in various ways:
  - Network sockets
  - Files
  - Named pipes (OS feature)

Blocking

- `MPI_Send` and `MPI_Recv` are both potentially *blocking* functions
- This means that every `MPI_Send` on one process must have a matching `MPI_Recv` on another process
  - **Or the process will hang**
- Messages are ’*nonovertaking*

Connection topologies

- Connecting multiple processing cores that are not on the same PC

Types of topologies:

- **Fully connected:** Every PC is connected to one another
- **Ring:** Every PC is connected to two other PCs to make a circle or a ring
- **Thin tree:** One computer at the top, for every level down, the number of PCs doubles, one connection each
- **Fat tree:** Same layout as thin tree however every level you go up the connections double
- **Torus:** A doughnut shape where every side is connected

Topology terminology

Diameter:

- The maximum path length between a pair of nodes
- High values of diameter cause more latency

Bisection width/Band width:

- The number of links you cut to divide the network into two equal halves
- To get the bisection bandwidth you multiply the bisection width with the bandwidth
- A high value is better as more links mean more resilience and high bandwidth helps with algorithm computation

Valency:

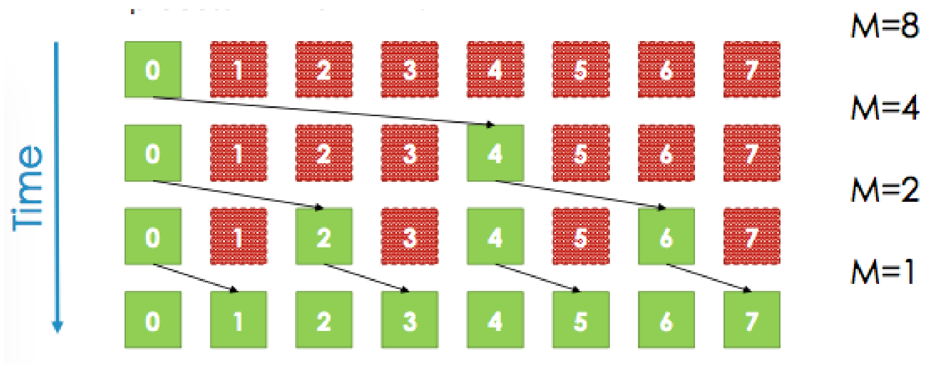
- How many connections each node makes
- Want to keep this to a reasonable amount
  - 6 or 7 max

Link count:

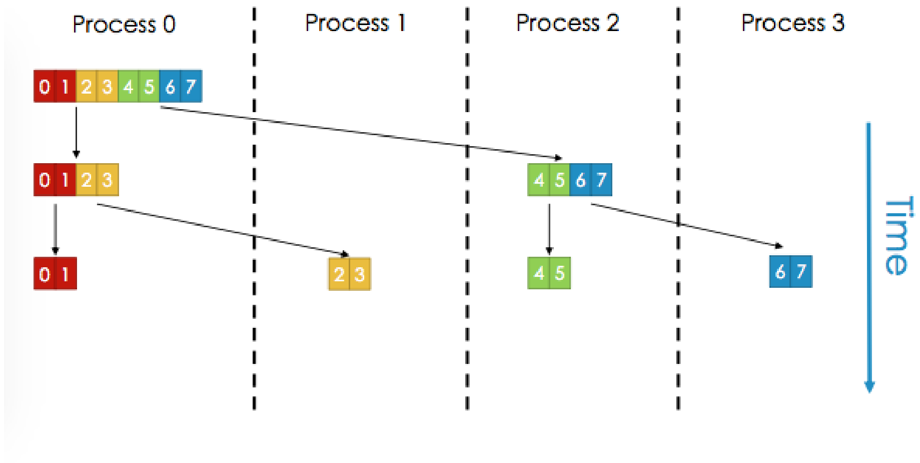
- How many connections the network has in total
- Shows how many wires are needed

Distributing data - binary tree

- Set `M` = the number of processes
- At every time step
  - Divide M by two
  - If a process with rank `i` has the data, it sends it to the process with rank `i + M`
- Very efficient at transmitting data
  - takes log2N time steps

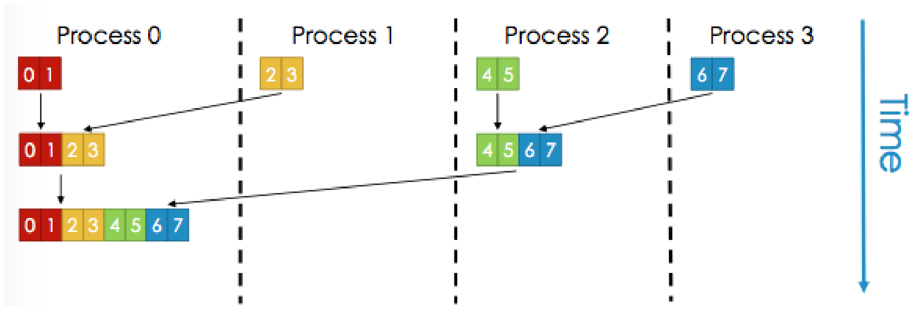


MPI_Scatter



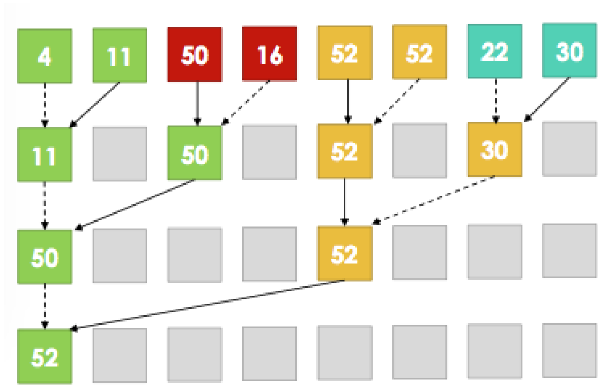
MPI_Gather

- Does the opposite of MPI_Scatter
- Collects data from arrays distributed across all the processes into one big array



MPI_Reduce

- Take an array of values and reduce it to a single value



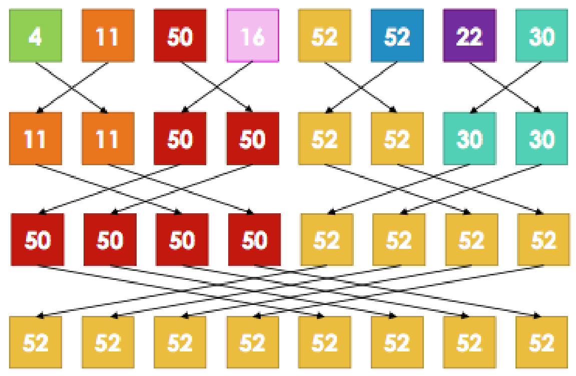
- Divide (active) processes into pairs.
- 2nd process in pair gives its value to 1st process, goes inactive.
- 1st process in pair takes the bigger of the two values.
- Repeat.

MPI_AllReduce

Pairs of individual processes compare values…

Pairs of blocks of 2 processes compare values…

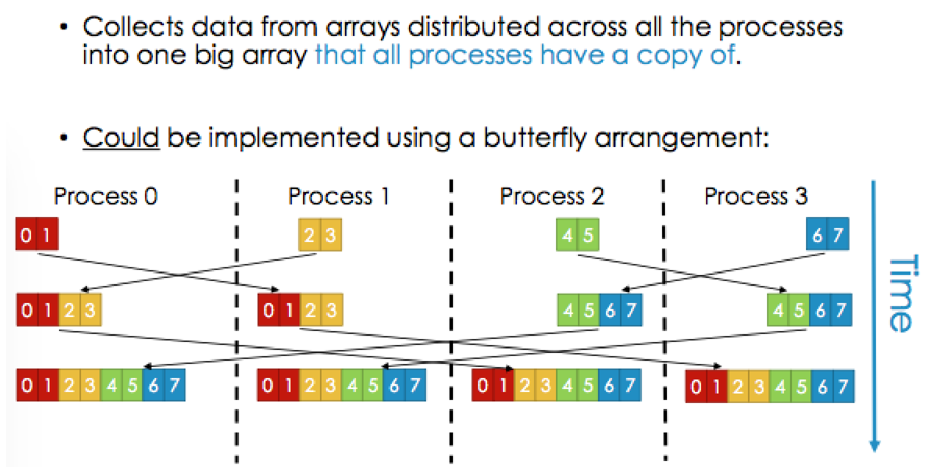
Pairs of blocks of 4 processes compare values…



Requires $\log_2 N$ steps – same as reduce or broadcast alone!

(But does require more data transfers…)

MPI_AllGather



- Collects data from arrays distributed across all the processes into one big array that all processes have a copy of.
- Could be implemented using a butterfly arrangement:

Topics that came up v topics that haven't come up

italic text means they've appeared twice

Have appeared	**Haven't appeared**
static allocation	Gustafon's law
—	—
pinned memory	what a SM is
—	—
which types of memory	difference between host and device
—	—
instruction level parallelism	difference bettwen static and dynamic
—	—
constant memory	disadvantages of performance with branching
—	—
how many threads will be created for a kernel launch	how threads can coporate to load data into shared mem
—	—
data organisation into cache	why CGMA is important
—	—
row/component major order	convolution
—	—
maximising occupancy	how shared memory can be optimised calculating a convolution

Have appeared	**Haven't appeared**
—	—
maximum number of blocks that can be resident in multiprocessor	difference between task and data parallelism
—	—
coalesced memory access	sublinear & super linear
—	—
CGMA	difference between threads and processes
—	—
syncthreads	`MPI_Send` & `MPI_Recv`
—	—
divergence	how `MPI` functions are blocking and non-overtaking
—	—
network topologies and their properties	different `MPI` collective communcation functions
—	—
register spilling and effect on other memory because of it	why and how collective communication functions can be efficient
—	—
CGMA of code	
—	—
SISD,SIMD,MISD,MIMD	
—	—
speedup	
—	—
efficieny	
—	—
amdahls	
—	—
linear speedup	
—	—
shared memory	
—	—
general formulas for topolgies	