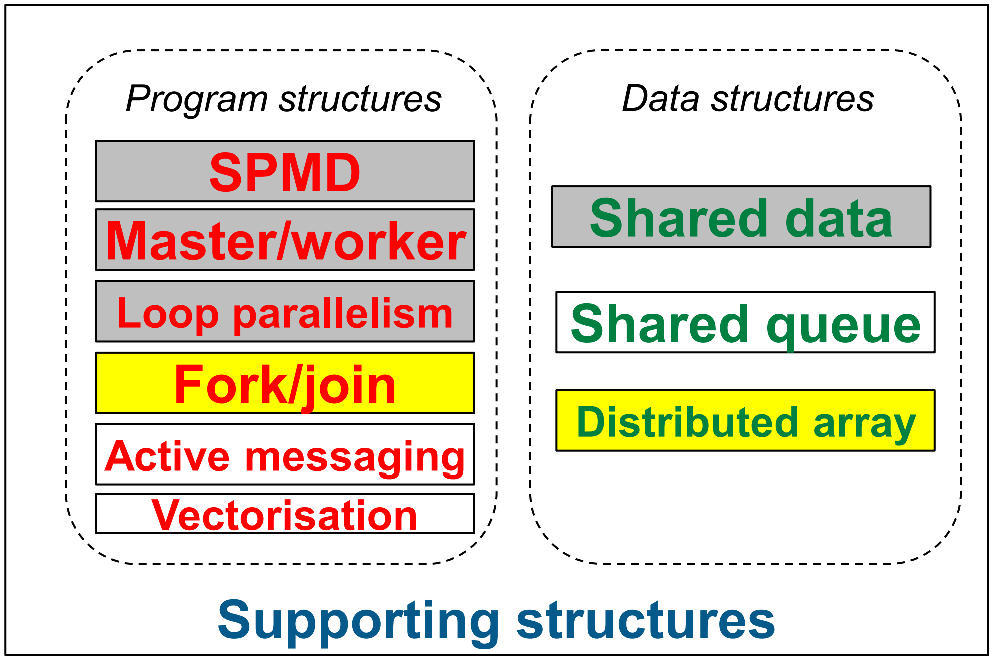
Parallel Design Patterns-L11

Distributed Arrays,

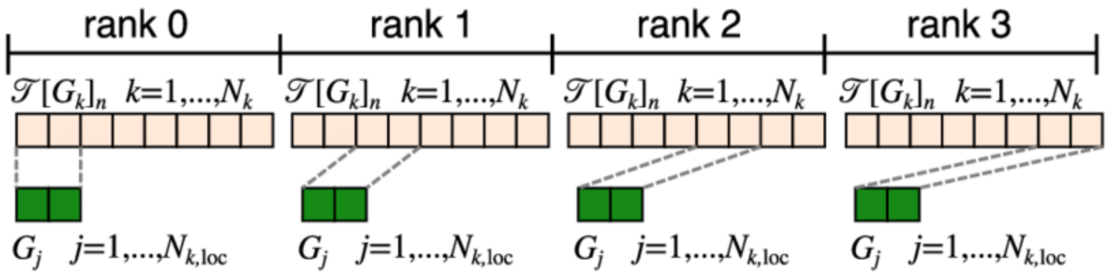
Fork Join

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes Room 2.13

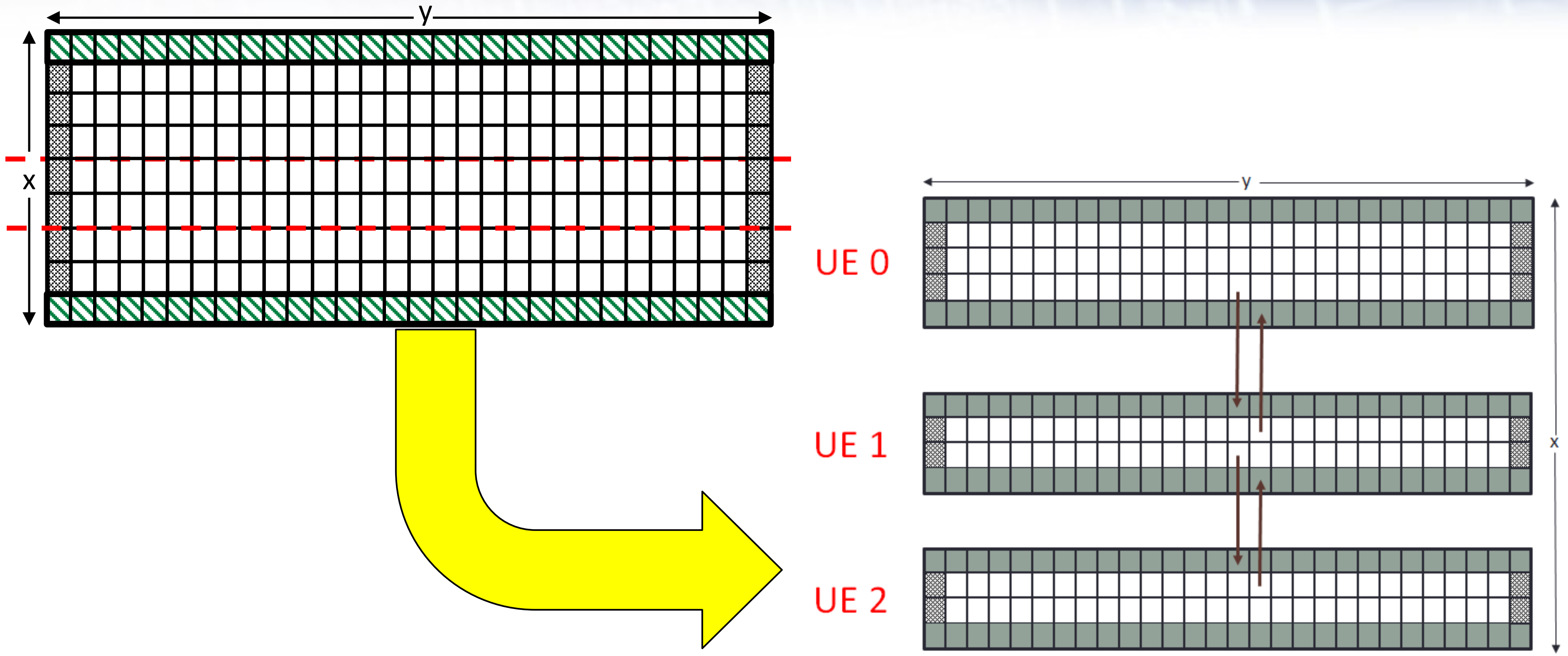

Program structures
SPMD
Master/worker
Loop parallelism
Fork/join
Active messaging
Vectorisation
Data structures
Shared data
Shared queue
Distributed array
Supporting structures

Distributed Array – Problem

- Large arrays are fundamental data structures in scientific computing problems.
- Most systems have memory access times that vary substantially depending on which UE is accessing a particular array element.
  - Even if that system supports a global address space
  - The challenge is to ensure that data elements are "*nearby*" at the right times during the computation
- For distributed systems, must explicitly distribute data.
- For NUMA desirable to have the right memory "*nearby*".


rank 0
rank 1
rank 2
rank 3
$\mathcal{T}[G_k]_n \quad k=1,\ldots,N_k$
$G_j \quad j=1,\ldots,N_{k,\mathrm{loc}}$

We have seen this already!



```
int local_nx=nx/size;
if (local_nx * size < nx) {
  if (myrank < nx - local_nx * size) local_nx++;
}
int mem_size_x=local_nx+2;
int mem_size_y=ny+2;
```

Distributed Array – Forces

- Load Balance
  - Aim to have a similar amount of work per UE
- Effective Memory Management
  - make good use of the cache and memory hierarchy
- Clarity of Solution
  - aim to have a clear mapping between local and global arrays
    - This can add complexity and be a source of bugs!

- The “*solution*” is the mapping between local and global arrays.
  - Typically our logical global view for the design of parallelism and local distribution for the concrete implementation details

Solution: with an 8×8 array example

$\lfloor(\cdots) \equiv floor(\cdots)$

$\lceil(\cdots) \equiv ceiling(\cdots)$

- Mapping an $M\times N$ matrix to P UEs...
  - element $a_{i,j}$ is assigned to p_k
  - Either 1D block or 1D block-cyclic
- Mapping an $M\times N$ matrix to $P\times Q$ UEs...
  - element $a_{i,j}$ is assigned to $p_{k,l}$
  - Either 2D block or 2D block-cyclic

M

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

N

1D Block with P = 4

$\lfloor(\cdots) \equiv floor(\cdots)$

$\lceil(\cdots) \equiv ceiling(\cdots)$

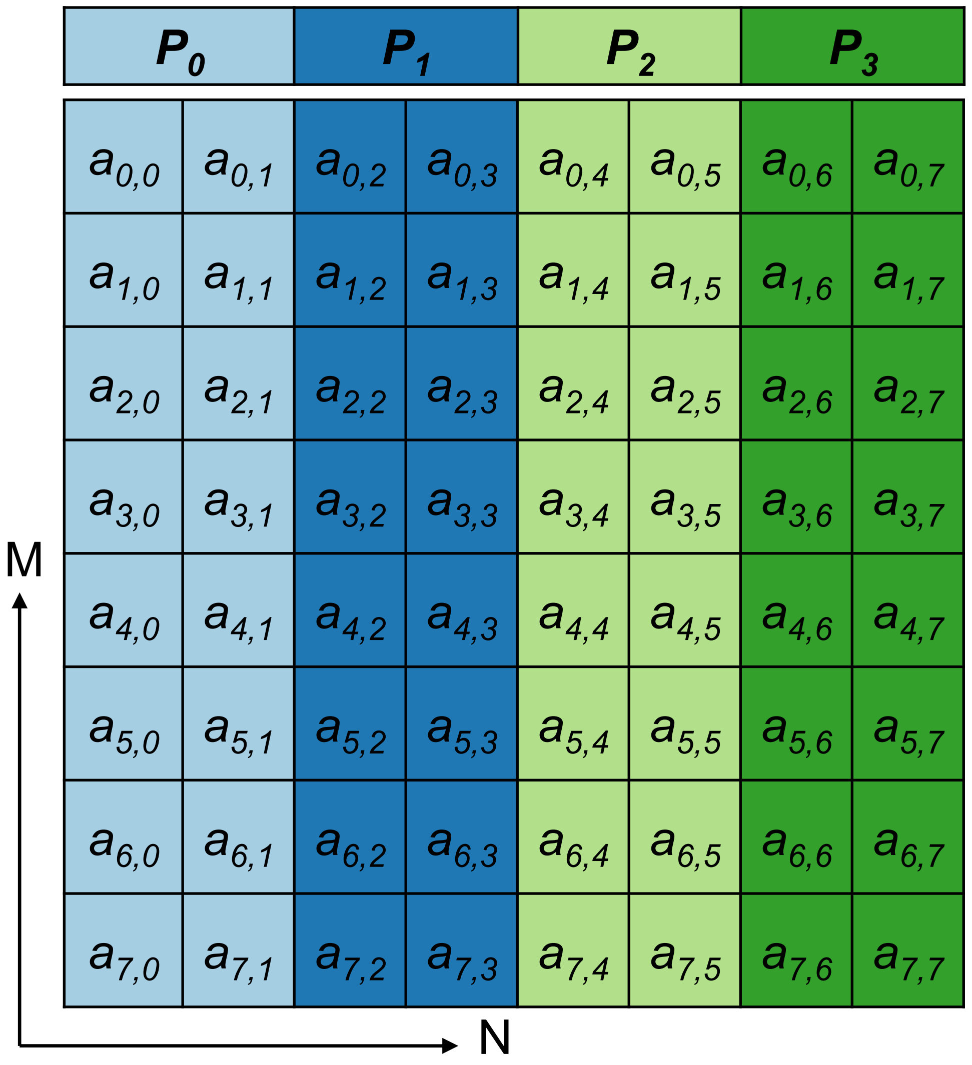
- Mapping an $M \times N$ matrix to P UEs...

$a_{i,j}$ assigned to p_k

$$k = \lfloor(j/\lceil(N/P))$$

$$j = [0..7]$$

$$M = 8$$



P_0		P_1		P_2		P_3	
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

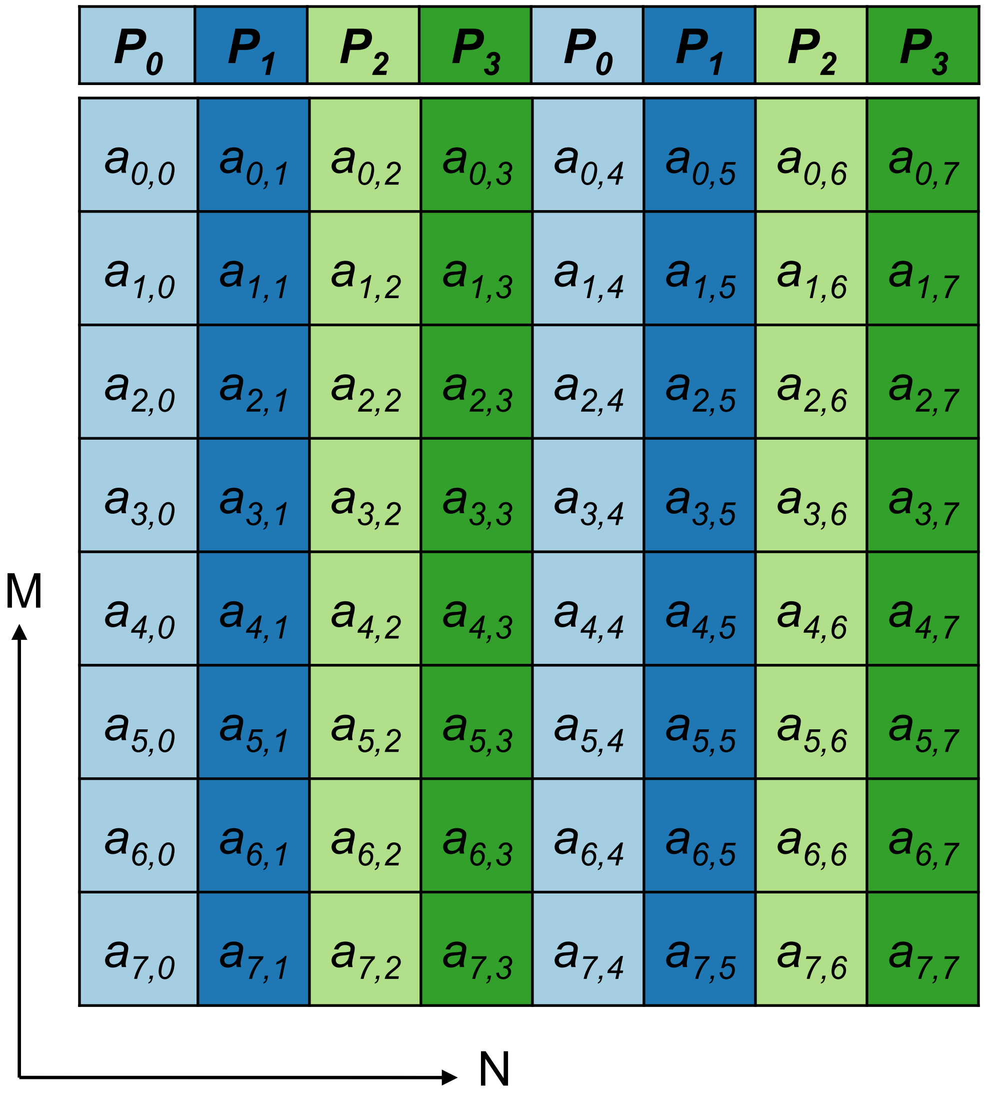
1D Block-cyclic with $P = 4$

- Mapping an $M \times N$ matrix to P UEs...

$a_{i,j}$ assigned to p_k

$$k = j \,\%\, P$$

$$j = [0..7]$$



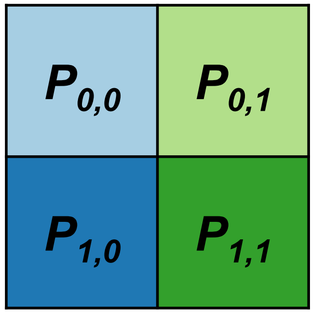
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

2D Block with P×Q = 2×2

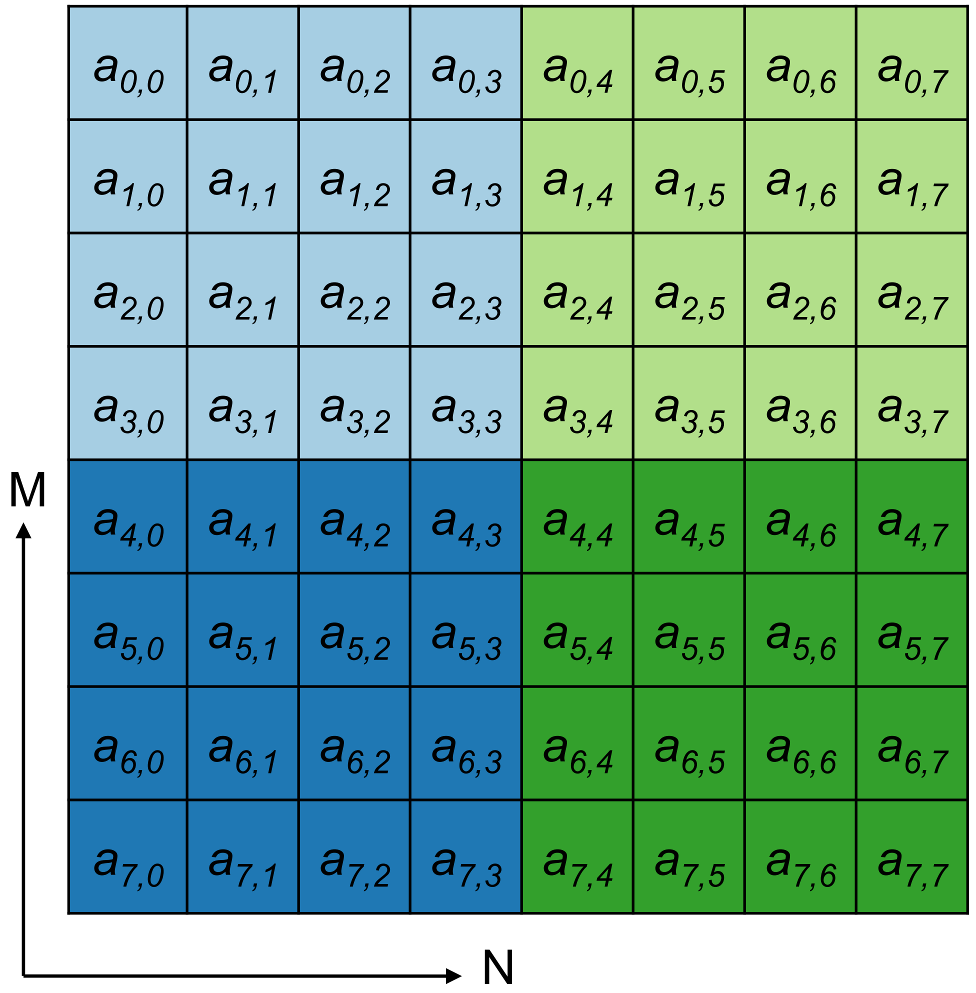
$$\lfloor(\cdots) \equiv floor(\cdots)$$
$$\lceil(\cdots) \equiv ceiling(\cdots)$$

$a_{i,j}$ assigned to $p_{k,l}$

$$k = \lfloor(i/\lceil(M/P))$$
$$l = \lfloor(j/\lceil(N/Q))$$
$$i,j = [0..7]$$
$$M = N = 8$$



$P_{0,0}$	$P_{0,1}$
$P_{1,0}$	$P_{1,1}$



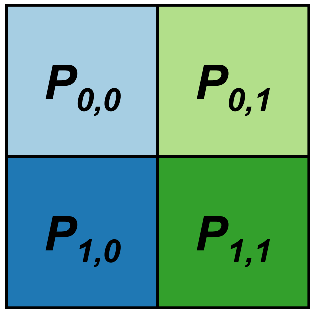
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

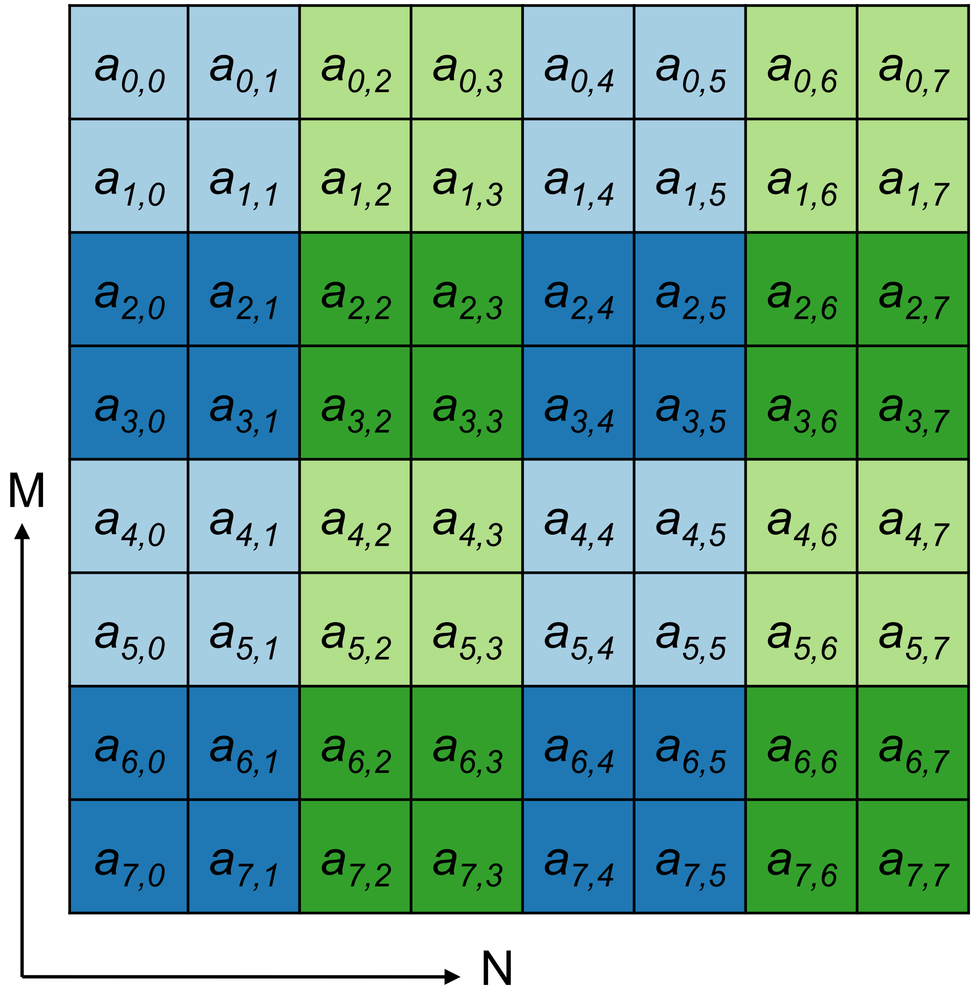
2D Block-cyclic with P×Q = 2×2

$$\lfloor(\cdots) \equiv floor(\cdots)$$
$$\lceil(\cdots) \equiv ceiling(\cdots)$$

$a_{i,j}$ assigned to $p_{k,l}$

$$k = \lfloor(i/\lceil(M/PQ)) \,\%\, P$$
$$l = \lfloor(j/\lceil(N/PQ)) \,\%\, Q$$
$$i,j = [0..7]$$
$$M = N = 8$$

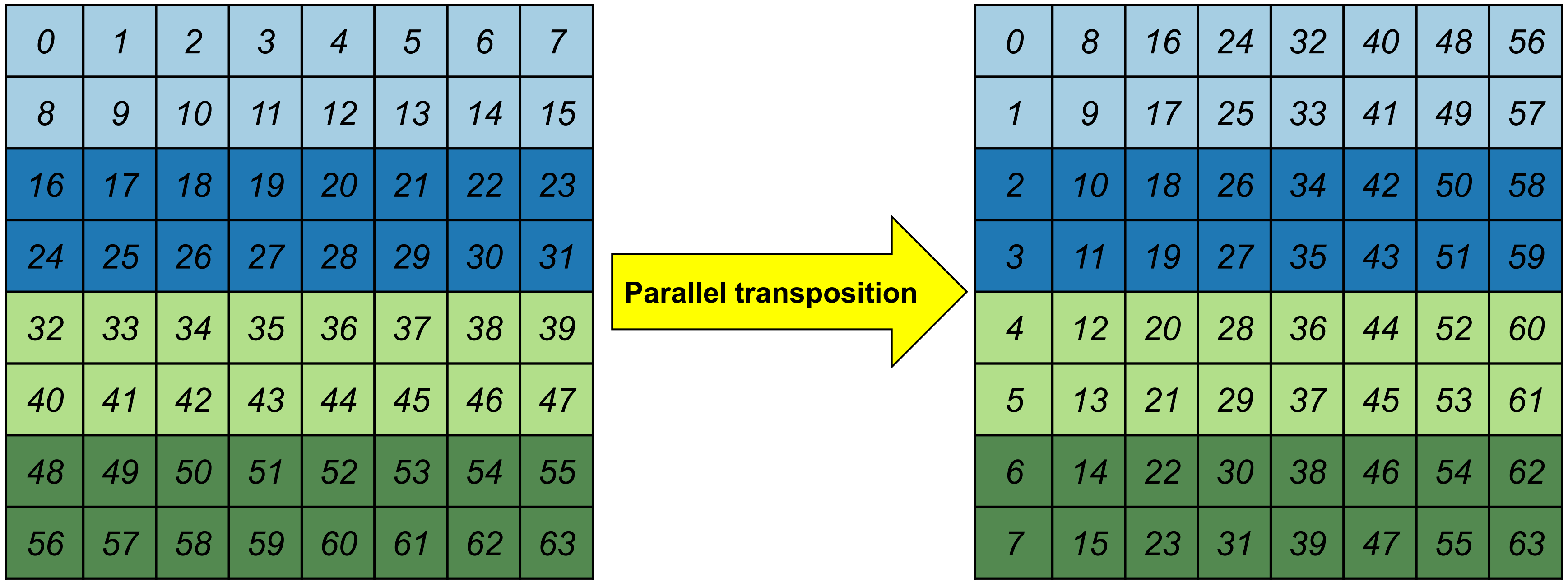




$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{3,6}$ $a_{3,7}$
$a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{4,6}$ $a_{4,7}$
$a_{5,0}$ $a_{5,1}$ $a_{5,2}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$ $a_{5,6}$ $a_{5,7}$
$a_{6,0}$ $a_{6,1}$ $a_{6,2}$ $a_{6,3}$ $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ $a_{6,7}$
$a_{7,0}$ $a_{7,1}$ $a_{7,2}$ $a_{7,3}$ $a_{7,4}$ $a_{7,5}$ $a_{7,6}$ $a_{7,7}$
M
N

Further distributed memory examples and uses

What makes parallel FFT difficult

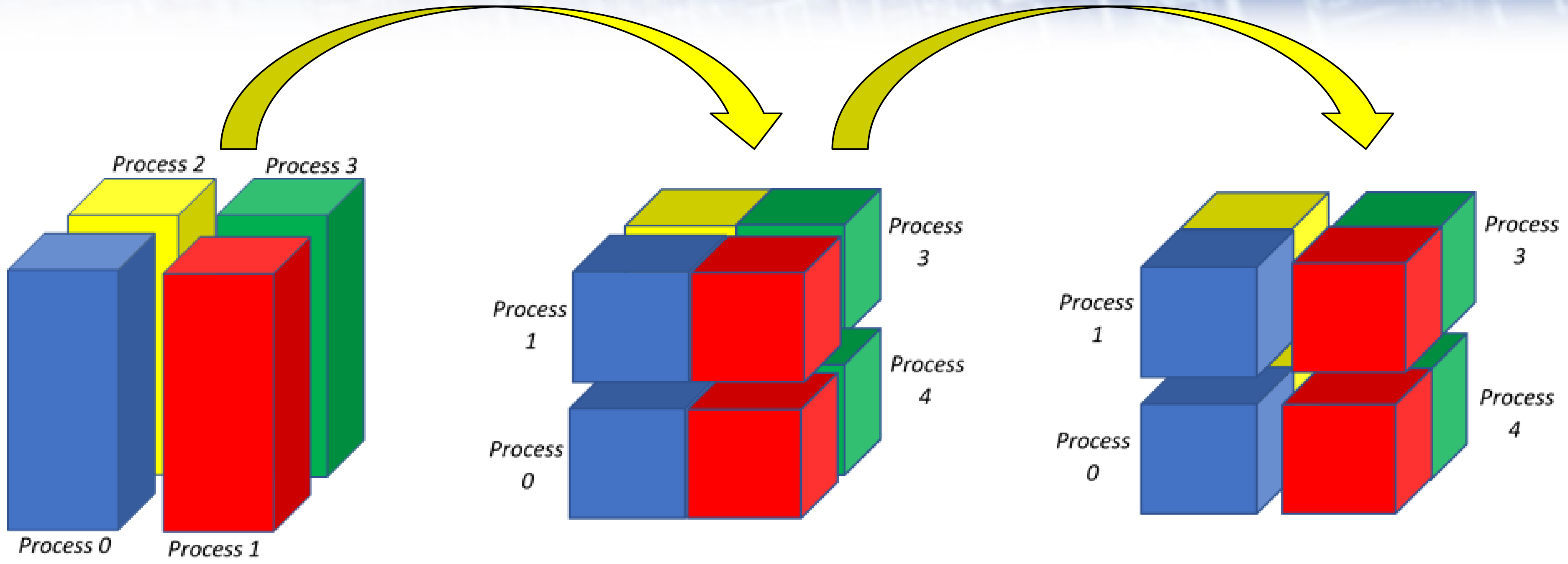


0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

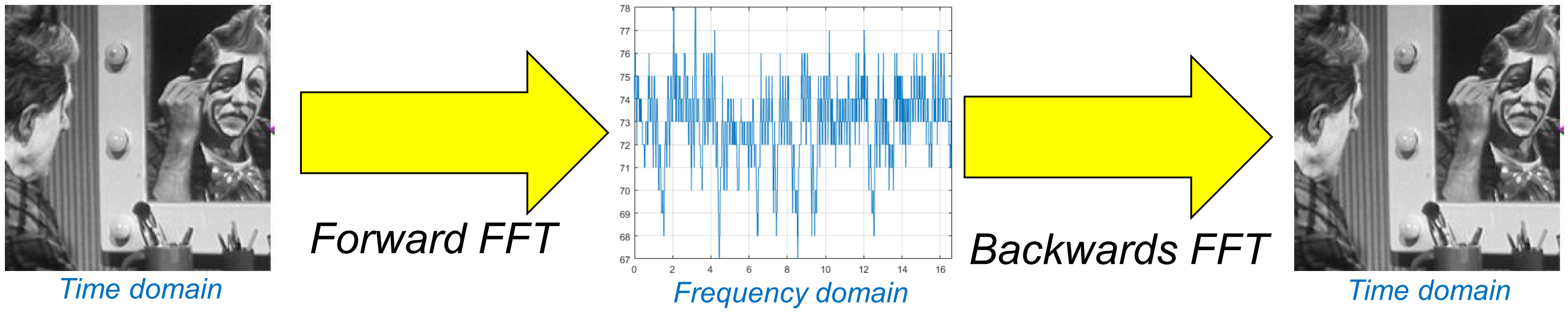
- Two dimensional data decomposed in one dimension
  - Parallel transposition needed, requiring all-to-all communications
    - Can add considerable overhead/poor scaling
  - Adds considerable complexity around edge cases, such as uneven decomposition of data
  - This 1D decomposition supported in FFTW for 2D and 3D data-sets

What makes parallel FFT difficult



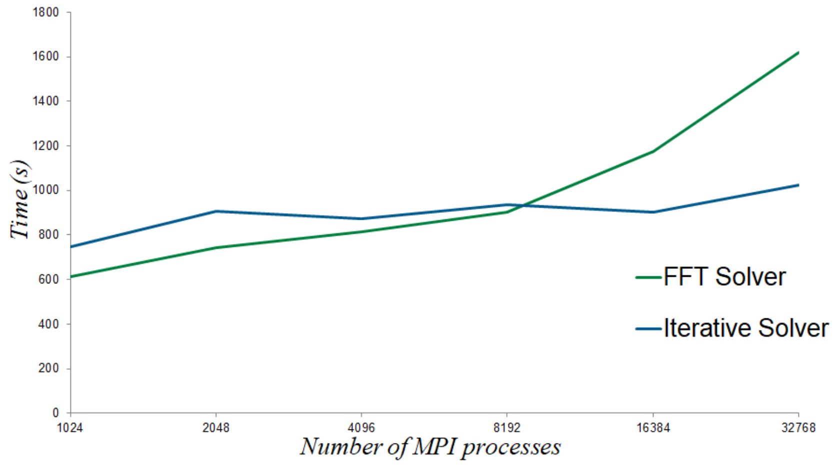
- Three dimensional data decomposed in two dimensions
  - Known as pencil decomposition
  - Parallel transposition now requires three steps and considerably more complex as need to communicate in different dimensions per step.
  - Also need to consider edge cases such as uneven decomposition of data but might only be for a specific step you need to hold this
  - Not supported by FFTW, although some third-party libraries have been developed for this

A property of FFTs that helps us out…

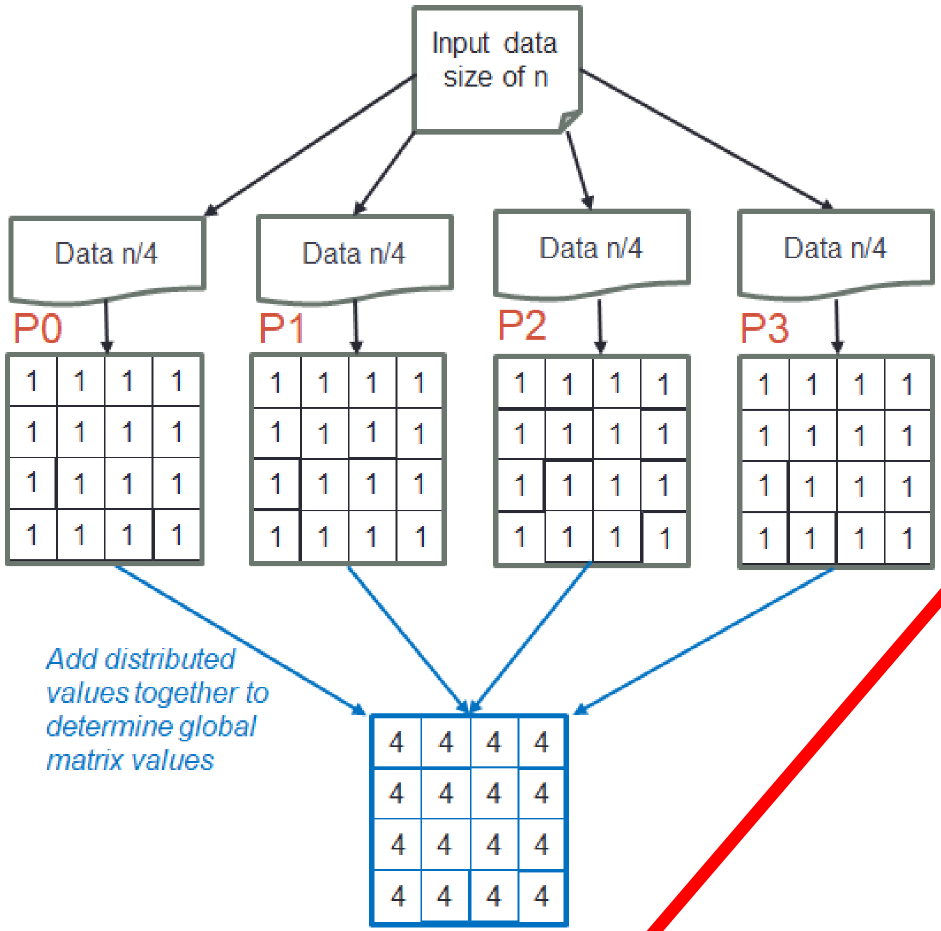


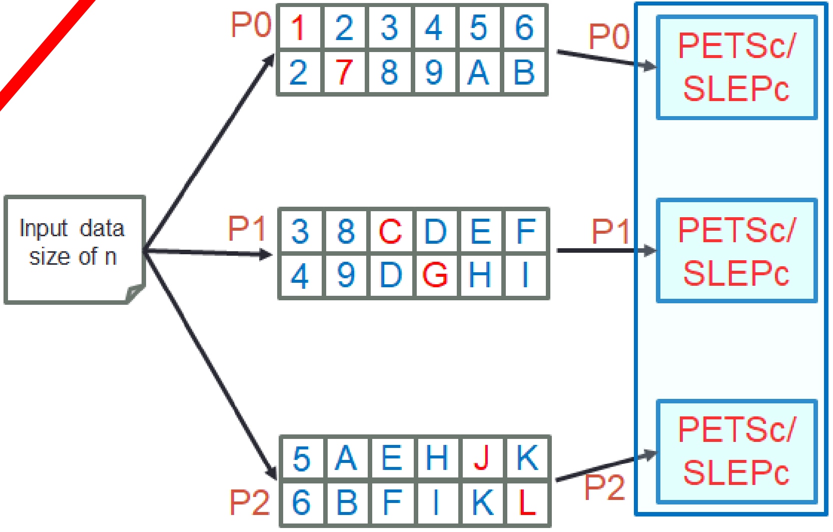
- If we do a forwards FFT and then a backwards FFT, the result should exactly match the input data
  - Can use this properly to easily test correctness of our parallel implementations

- Parallel FFT does involve all-to-all communications, but can still scale at smaller numbers of UEs

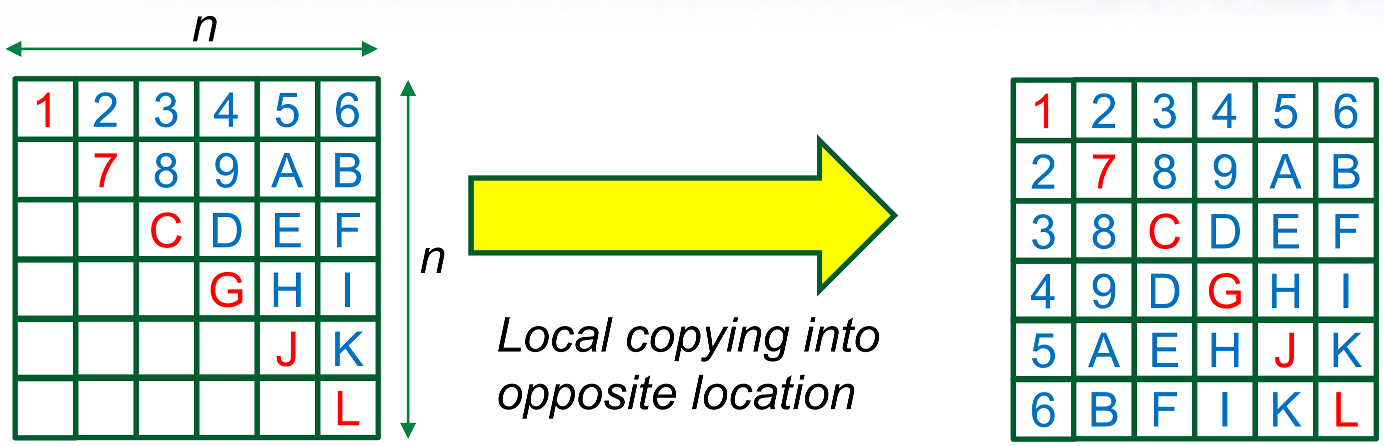


Distributed Array: Example

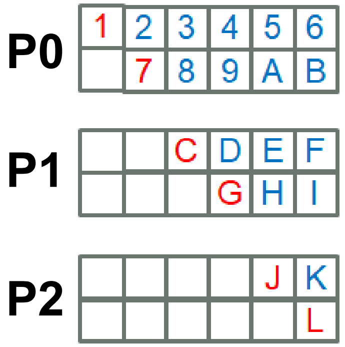




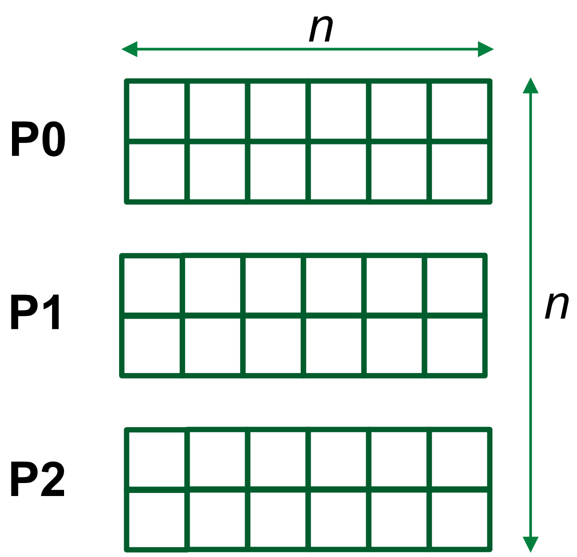
Distributed Array: Example



- With entirety of matrix on each process, the symmetry is simple to deal with as only compute the diagonal and upper part, and copy upper elements into lower locations
  - When we split the matrix up could just calculate upper elements and communicate to the lower elements
    - But significant load imbalance!
  - Or could compute all elements, but duplication of work



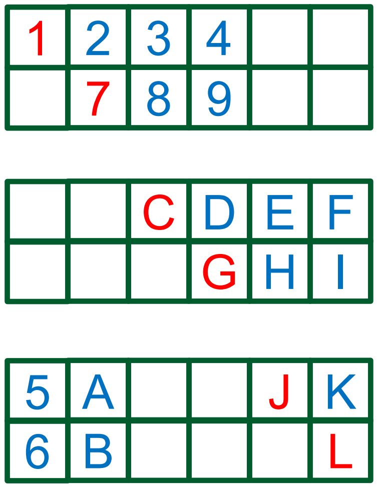
Distributed Array: Example



- Total number of points to be explicitly calculated $$f = \frac{n^2 - n}{2} + n$$
- Base points per row to be explicitly calculated $$r = \frac{f}{n}$$

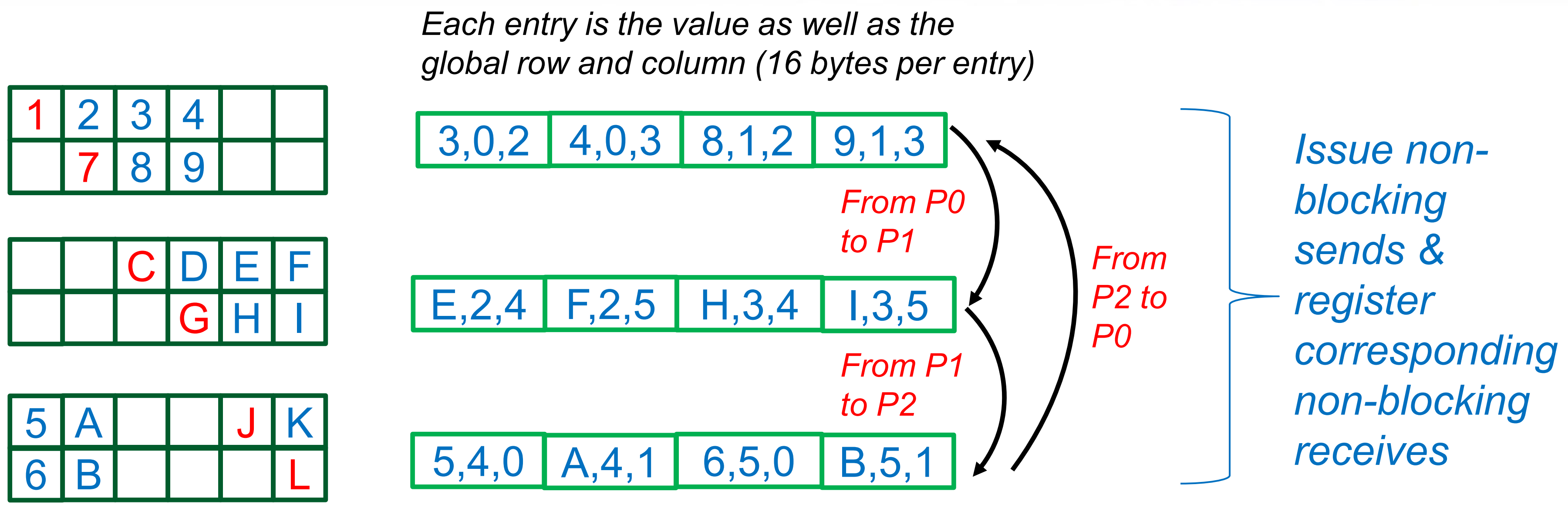
f=21

r=3.5



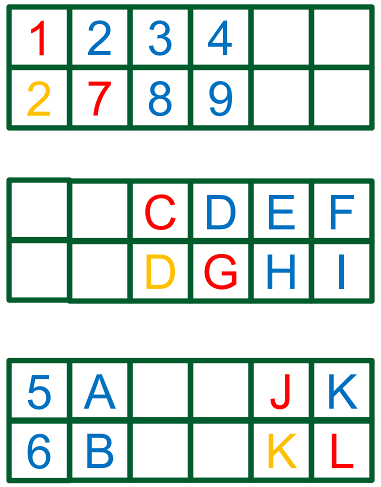
- Starting at the diagonal, start calculating r local points.
  - If r is fractional (n is even), alternate between ceil(r) and floor(r) points for each row

Distributed Array: Example

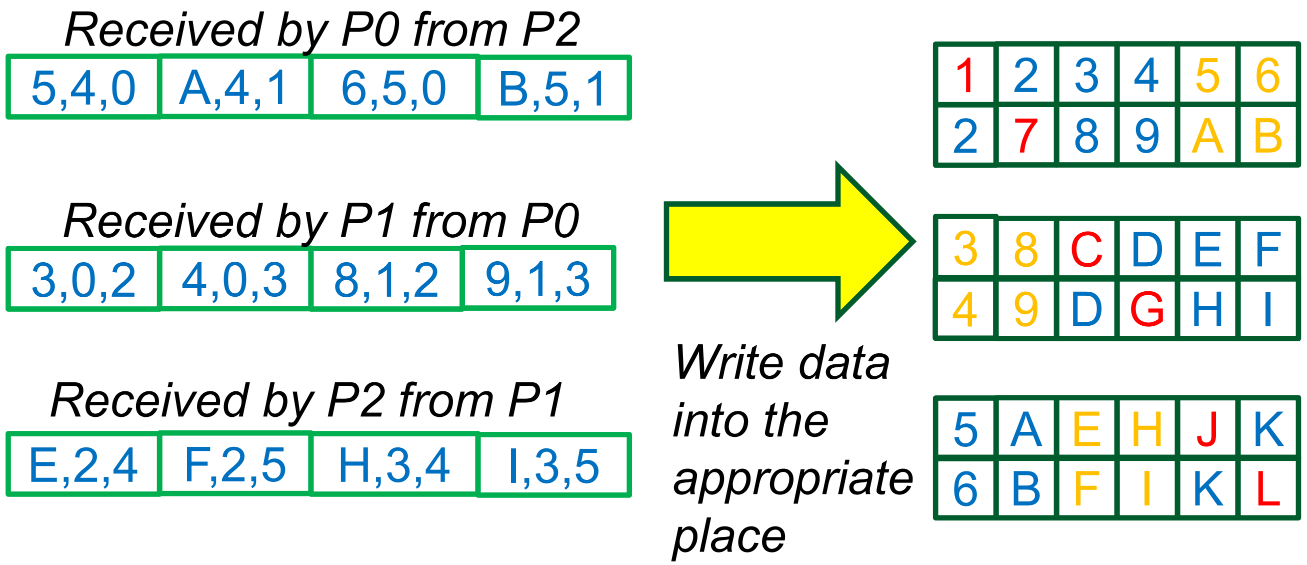


Each entry is the value as well as the global row and column (16 bytes per entry)
1 2 3 4
7 8 9
C D E F
G H I
5 A J K
6 B L
3,0,2 4,0,3 8,1,2 9,1,3
From P0 to P1
E,2,4 F,2,5 H,3,4 I,3,5
From P1 to P2
5,4,0 A,4,1 6,5,0 B,5,1
From P2 to P0
Issue non-blocking sends & register corresponding non-blocking receives

- Next we copy all local values (between locally held rows)
- Once we have done this wait for all communications to complete
  - Overlapping the local data copy with the communications



Distributed Array: Example



Received by P0 from P2

5,4,0	A,4,1	6,5,0	B,5,1

Received by P1 from P0

3,0,2	4,0,3	8,1,2	9,1,3

Received by P2 from P1

E,2,4	F,2,5	H,3,4	I,3,5

Write data into the appropriate place

1	2	3	4	5	6
2	7	8	9	A	B

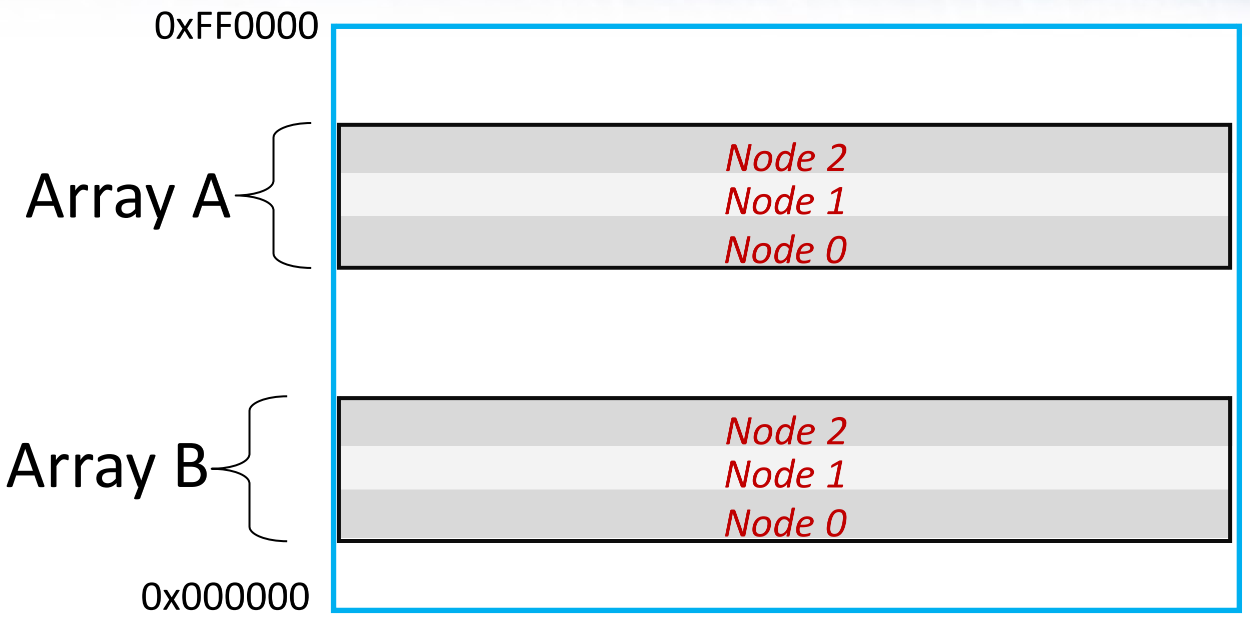
3	8	C	D	E	F
4	9	D	G	H	I

5	A	E	H	J	K
6	B	F	I	K	L

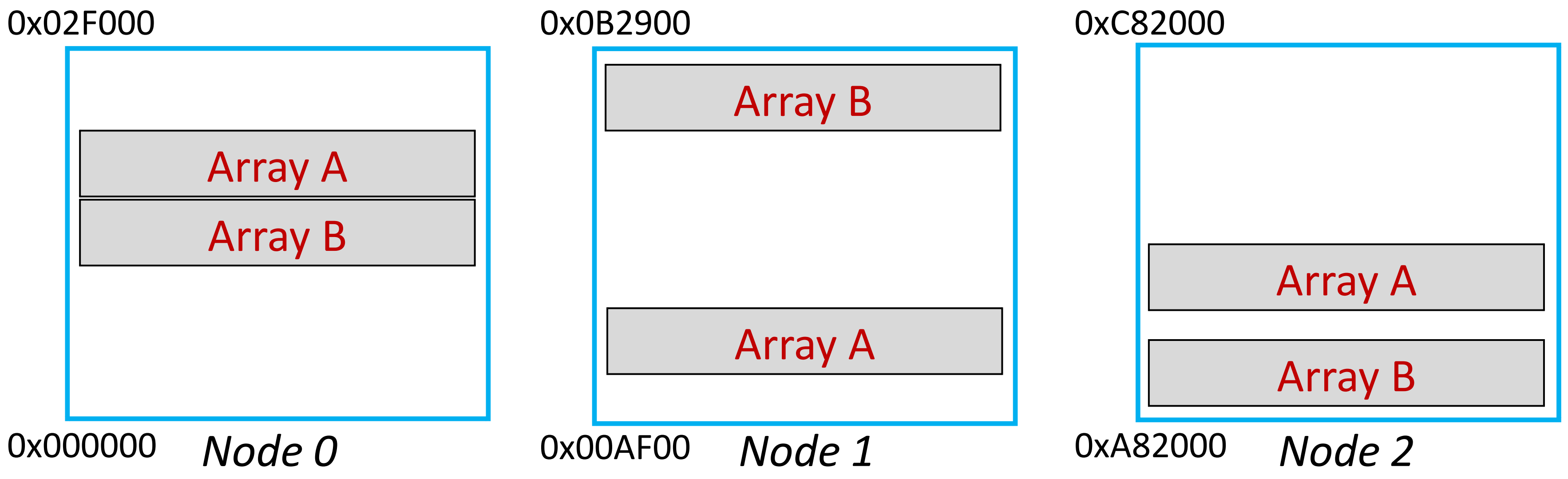
As each received data value also has associated its global row and column, it is trivial to place it in the appropriate location

- We still need communication of the values, but don’t need communication to coordinate which processes calculate what
  - At worst each process needs to communicate with every other process, but this is 1 single large message
  - Including indexes in receive message is maybe not ideal at first glance, but works well and saves complexity on receiving end

In the research space: Directory/cache



- Provide a global view of the data, by exploiting the virtual address space of the CPU
  - If no physical page will raise an interrupt on access, can intercept this and undertake required data movement



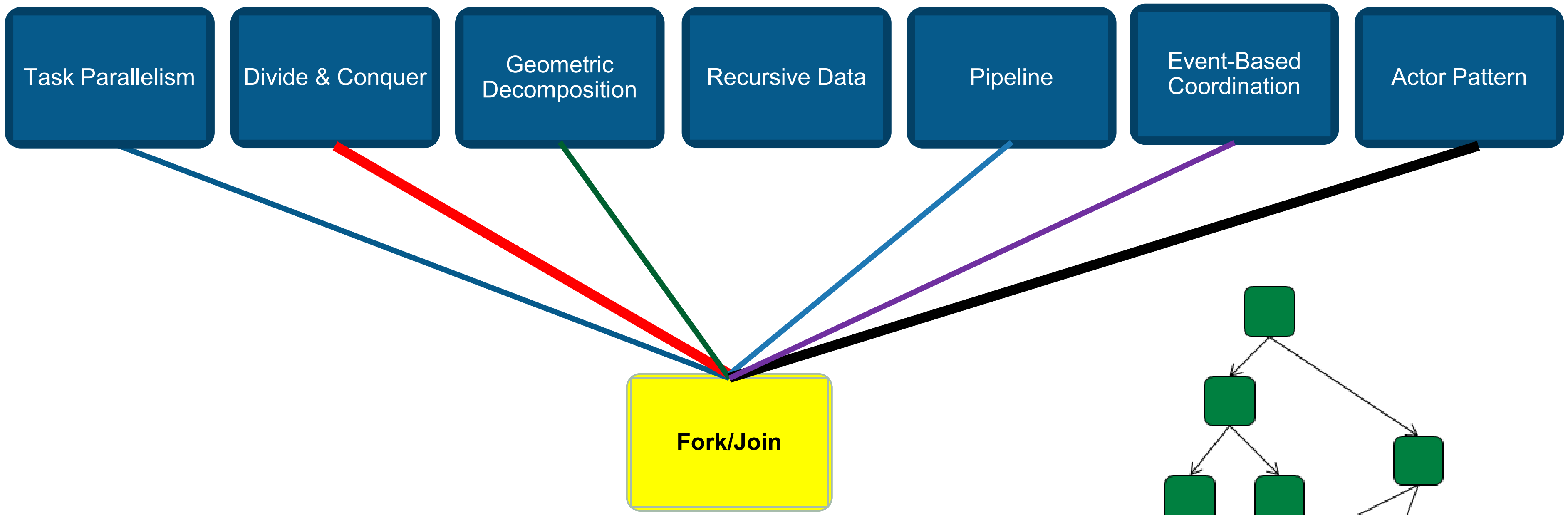
Distributed Array – Comments

- Complex mappings between co-ordinate systems are often best-expressed by use of macros.
  - Aids readability and harder to make mistakes when writing
  - No performance hit as expanded at compile time

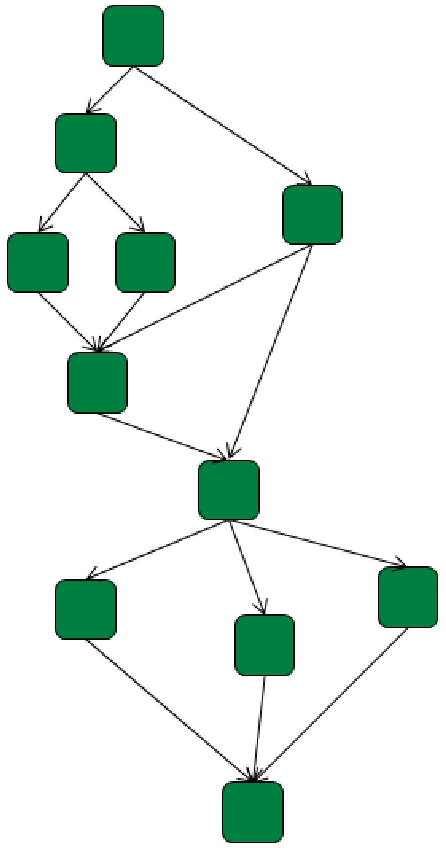
- ScaLAPACK is an example of a library that is based around the 2D block-cyclic array distribution
  - good for load balance and memory locality
    - http://netlib.org/scalapack/slug/node75.html

- Distributed Array is often used with the Geometric Decomposition and SPMD patterns.

Fork/Join



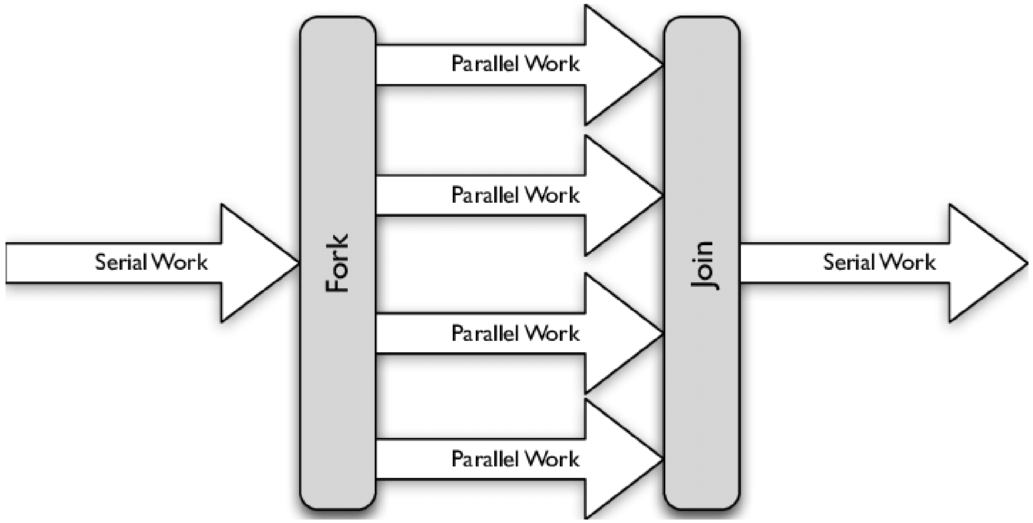
- The number of concurrent tasks varies throughout the execution of the program and a simple control structure such as a parallel loop is not sufficient. How can a parallel program be constructed around these dynamic set of tasks?



Fork/Join: The Context

- Applicable where the algorithm imposes an irregular or dynamic control structure
  - Tasks are created dynamically (*forked*) and terminated (*joined* with the forking task) as the program continues to execute
    - Although this forking might only happen once, at program start, and join at the end of the program
  - Loop parallelism can be thought of as a special case of Fork-Join where the forking pattern is very regular

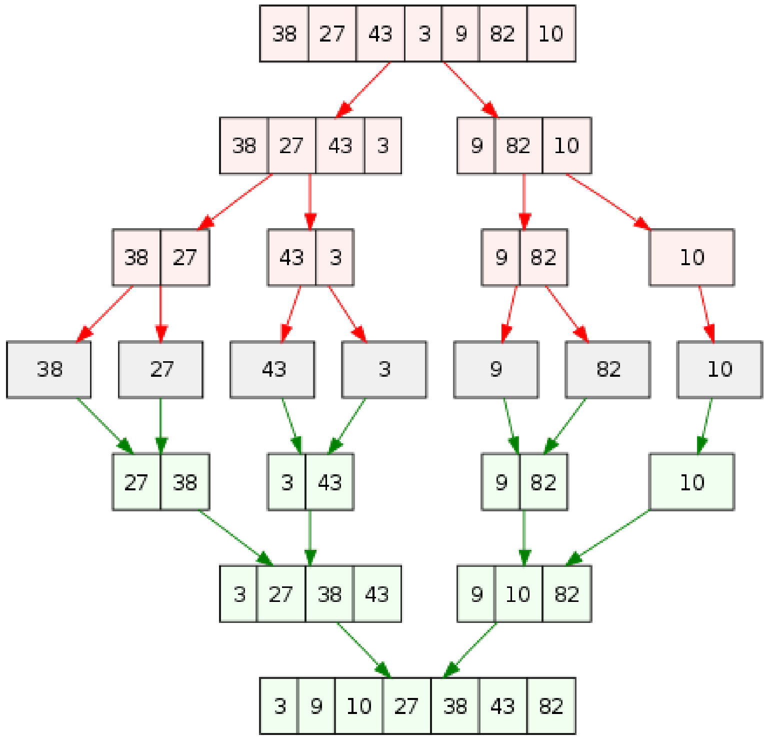
- A good match, for example, with the divide & conquer pattern discussed previously



Fork-Join: The Forces

- Algorithms often imply relationships between tasks, with the relationships arise dynamically. It can be useful to have the relationship between the UEs closely match the relationship between the tasks

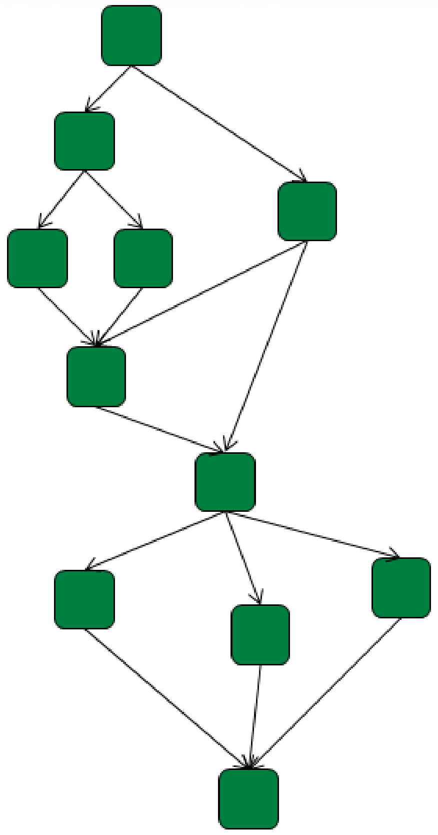
- A one-to-one mapping between UEs and tasks is usually natural
  - But must be balanced against the number of UEs provided
  - UE creation and destruction are expensive operations. It may be desirable to structure the program so as to restrict the number of forks and joins.



Fork/Join: The Solution

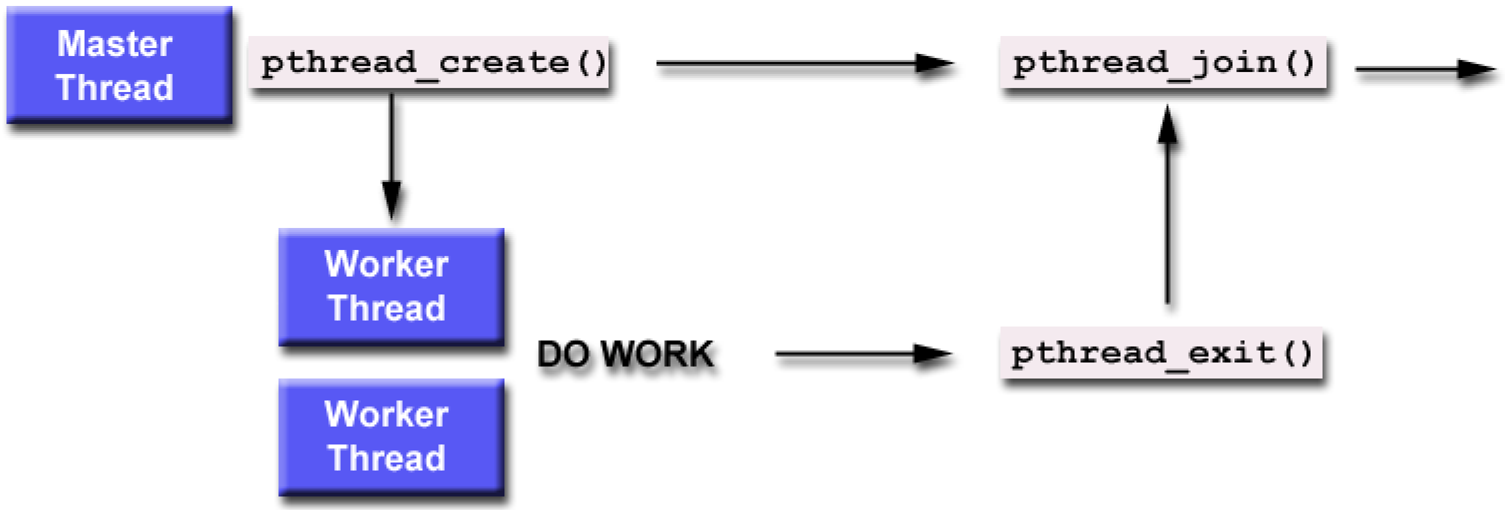
- Two Possible Solutions:
  - Direct task/UE mapping
  - Indirect task/UE mapping

- With Fork/Join the UEs are usually (but don't have to be) threads



- In both cases, a fork results in an extra task (or several extra tasks) and a join results in the completion of these tasks
  - Whether these map directly to the UEs (threads) depends on direct or indirect mapping

Direct Mapping



The simplest case:

- Map each task to a single UE
- As new tasks are created, new UEs are created
- There is almost always a synchronisation point where the parent (forking) UE waits for the forked tasks to complete and the forked UE to re-join

Direct mapping Fork/Join in OpenMP

```
#pragma omp parallel sections
{
  #pragma omp section
  {
     ......
  }
  #pragma omp section
  {
     ......
  }
}
```

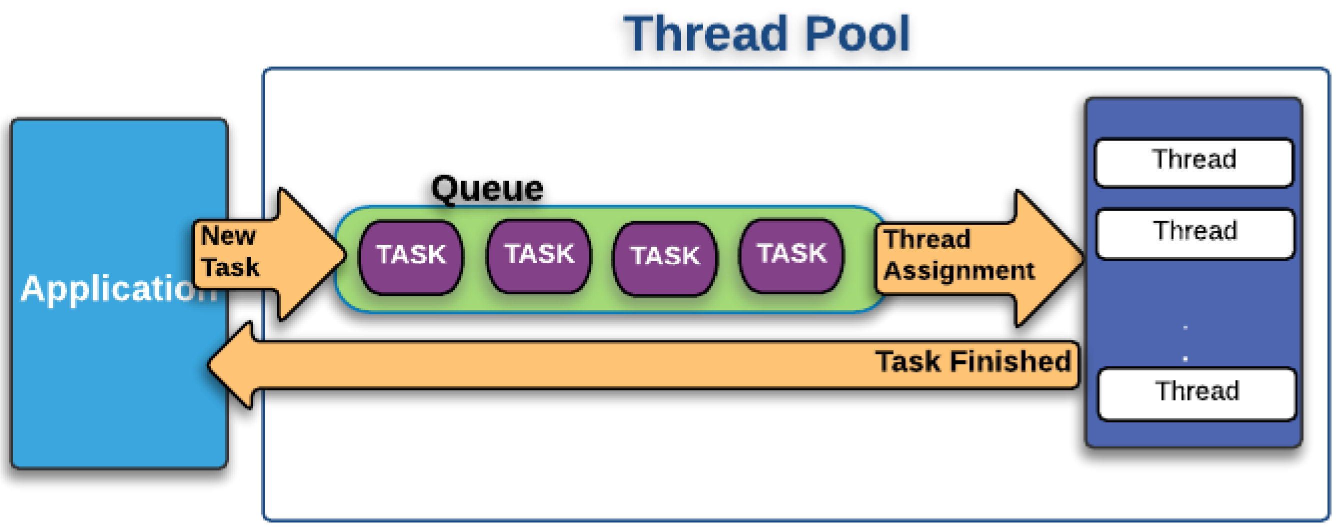
Fork - one thread executes code in here

Fork - one thread executes code in here

Join - all threads block here

- An example of non iterative loop directives in OpenMP
- Sections are directly mapping the contained code (the task) to a thread in the parallel region’s team
  - Limitation that can not nest sections due to direct mapping

Indirect Mapping



- Use a thread pool
- Create threads at the start
  - usually with same number of UEs as PEs
  - Cheaper than repeated thread creation/destruction
- Forking corresponds to putting a task into the thread pool, and join to sending results back
- Similar to low-level implementation of Master-Worker pattern

Indirect mapping Fork/Join in OpenMP

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
       { some task }
    #pragma omp task
       { some task }
    #pragma omp task
       { some task }
    #pragma omp taskwait
  }
}
```

Fork – enqueue a task to be executed by a thread at some point

Fork – enqueue a task to be executed by a thread at some point

Fork – enqueue a task to be executed by a thread at some point

Join – wait for all child tasks to complete

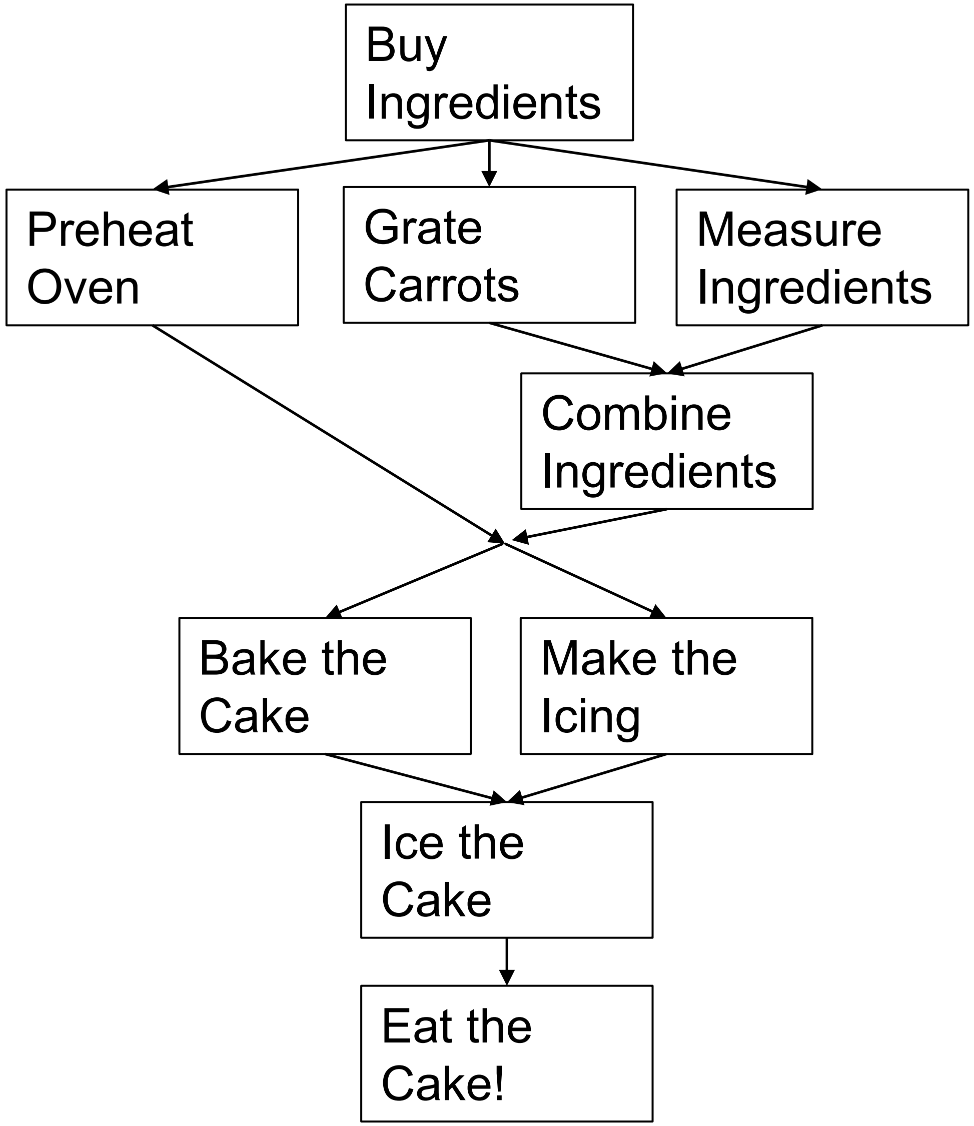
- OpenMP will queue up tasks and run them *at some point* on a thread in the parallel region
  - See how only a single thread is scheduling these
  - Taskwait will join, can be more complex than direct mapping

Fork-Join: OpenMP, Java, MPI

- The Fork/Join pattern is the standard programming model in OpenMP
  - OpenMP programs start as a single thread and on reaching a parallel region, a team of threads is forked
  - At the end of the parallel region, the threads rejoin their parent
  - In the case of loops, you get the special *loop parallelism* case
  - Therefore loop parallelism pattern is built upon fork/join pattern
- Can be implemented with MPI, but not so natural
  - In this case, indirect mapping / process pools are often used
- The Fork/Join pattern is also the standard implementation model for Java threads and pthreads as well as many others

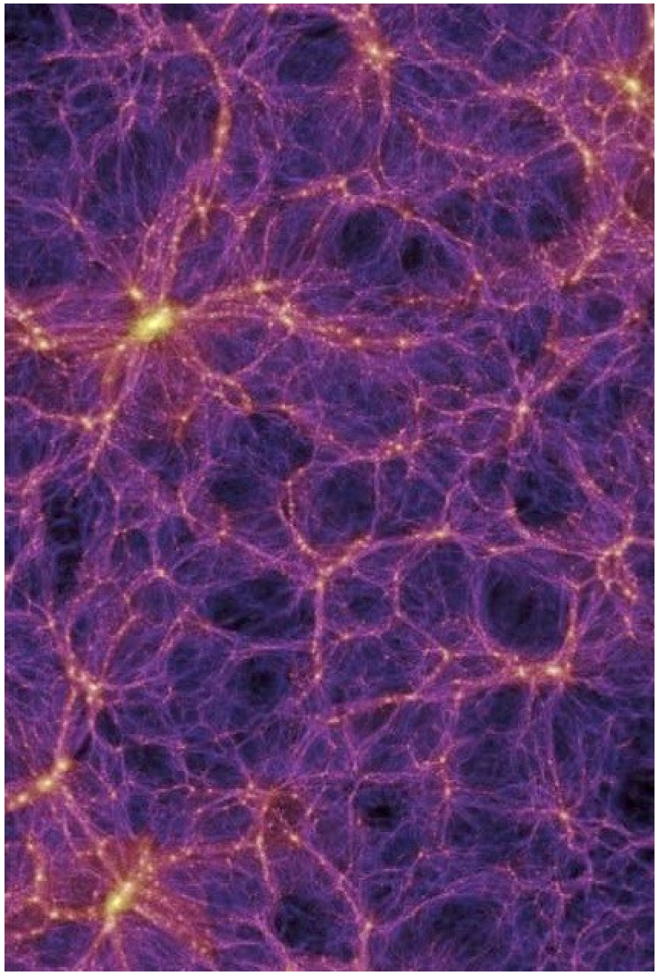
Fork/Join example: Baking a Carrot Cake

1. Buy ingredients
2. Preheat oven
3. Grate the carrots
4. Measure out ingredients
5. Combine ingredients
6. Bake the cake
7. Make the icing
8. Ice the cake
9. Eat



Fork/Join example: Tree Construction

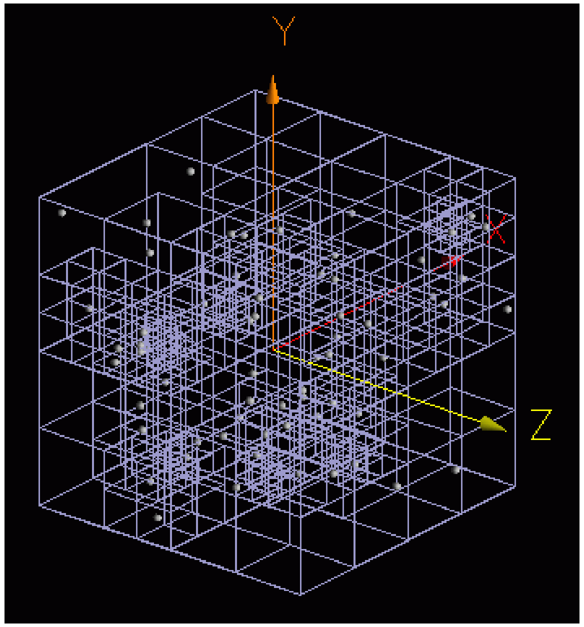
- In N-body gravitational simulations, the force on a given particle is the sum of the forces from each of the other N-1 particles
- Each particle must have its force evaluated, resulting in ~O(N^2) force evaluations per time step
- But, for particles far away from each other, do we really need to evaluate these forces accurately?



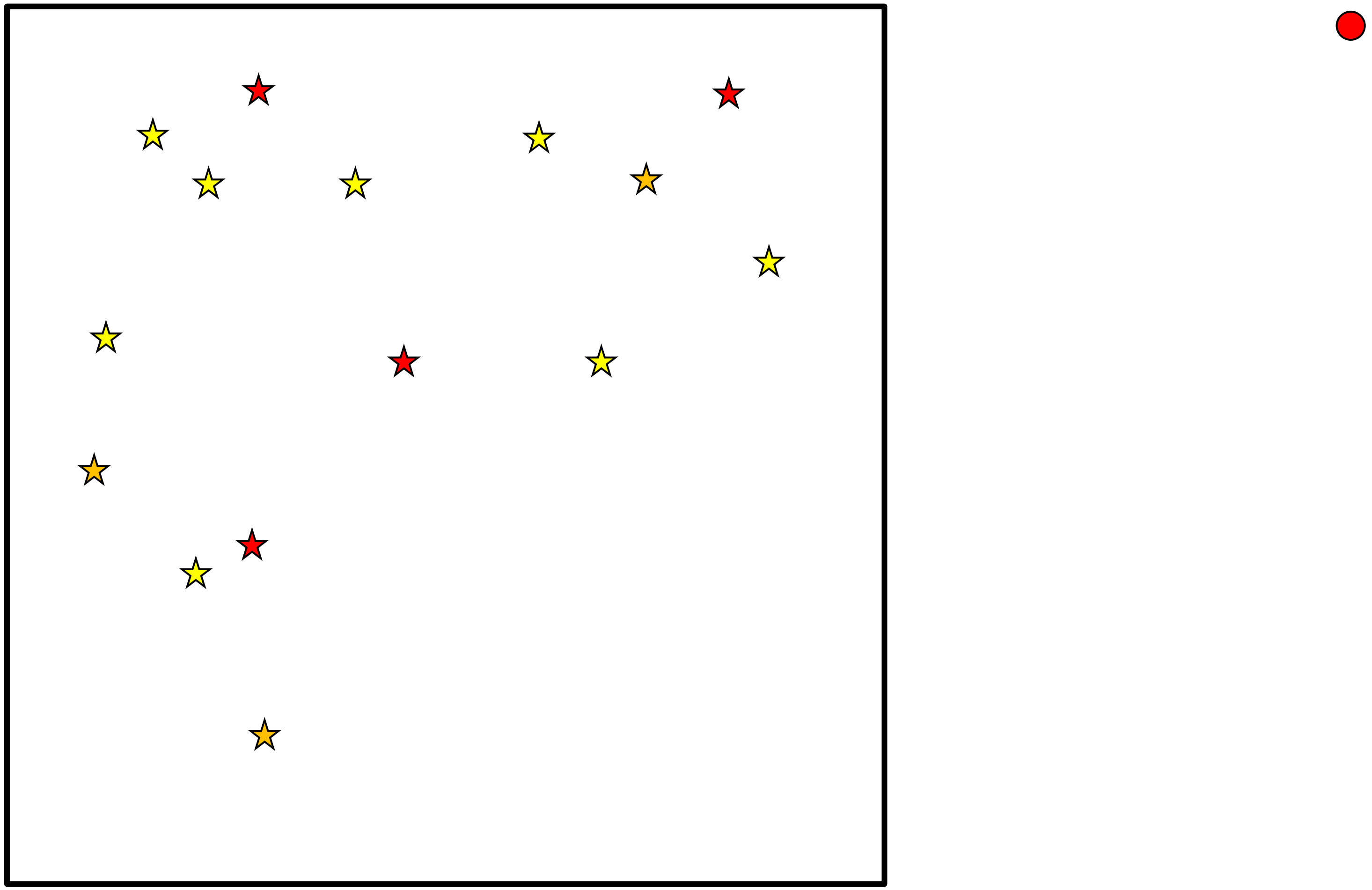
Barnes Hut Tree

One way to approximate the forces from distant particles is through the use of a Barnes Hut tree:

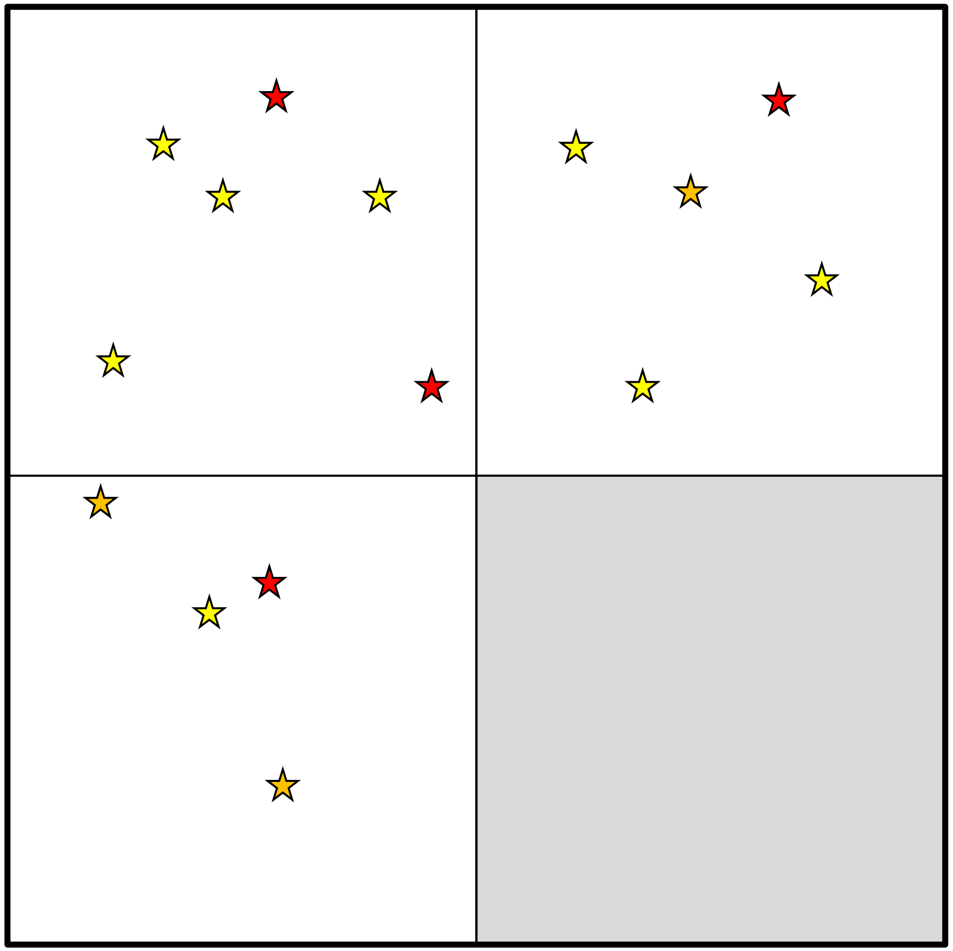
- First, divide the domain into 4 quadrants (2-D) or 8 octants (3-D) (also known as ‘nodes’)
- If a quadrant has 1 or 0 stars in it, we’re done
- If it has more than one star in it, calculate (and store) its centre of mass and total mass, then subdivide the quadrant again
- Repeat until every quadrant contains 1 or 0 stars

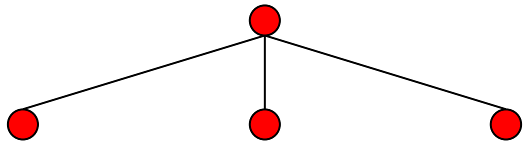


Barnes Hut Tree - Construction

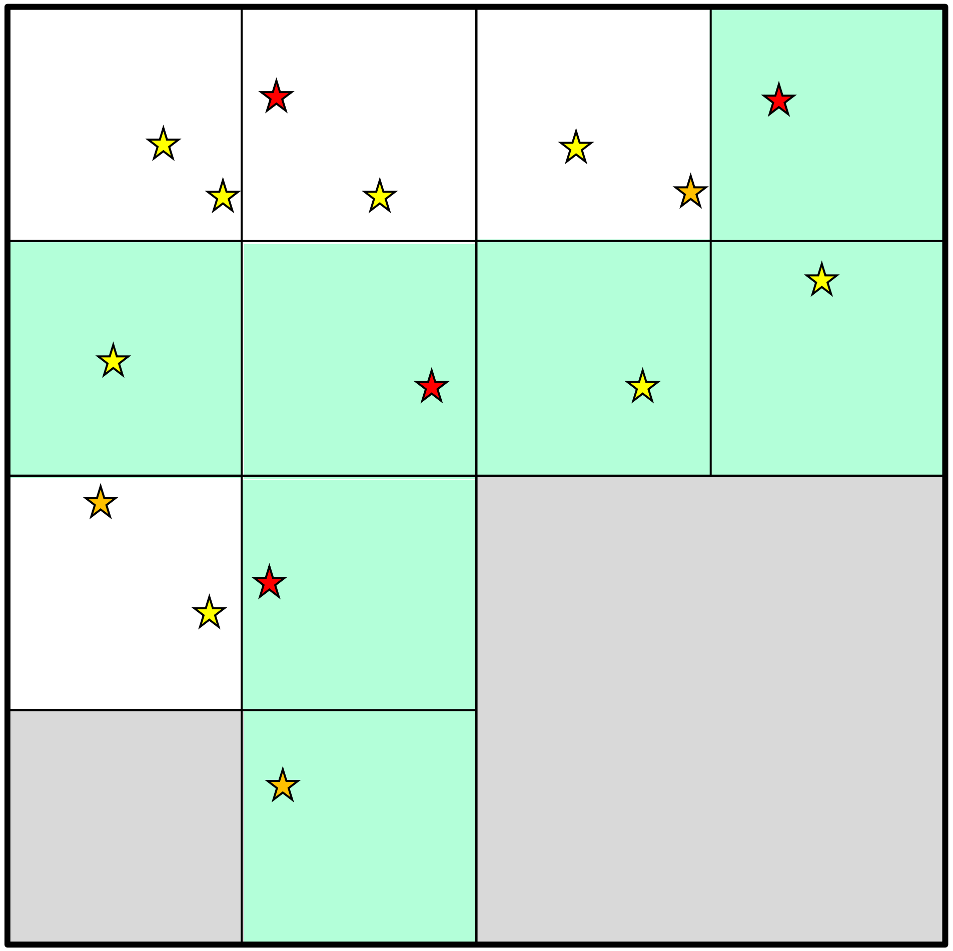


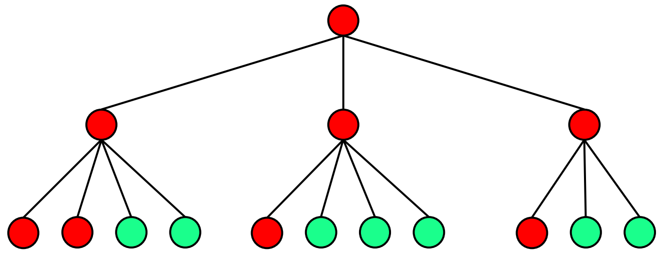
Barnes Hut Tree - Construction





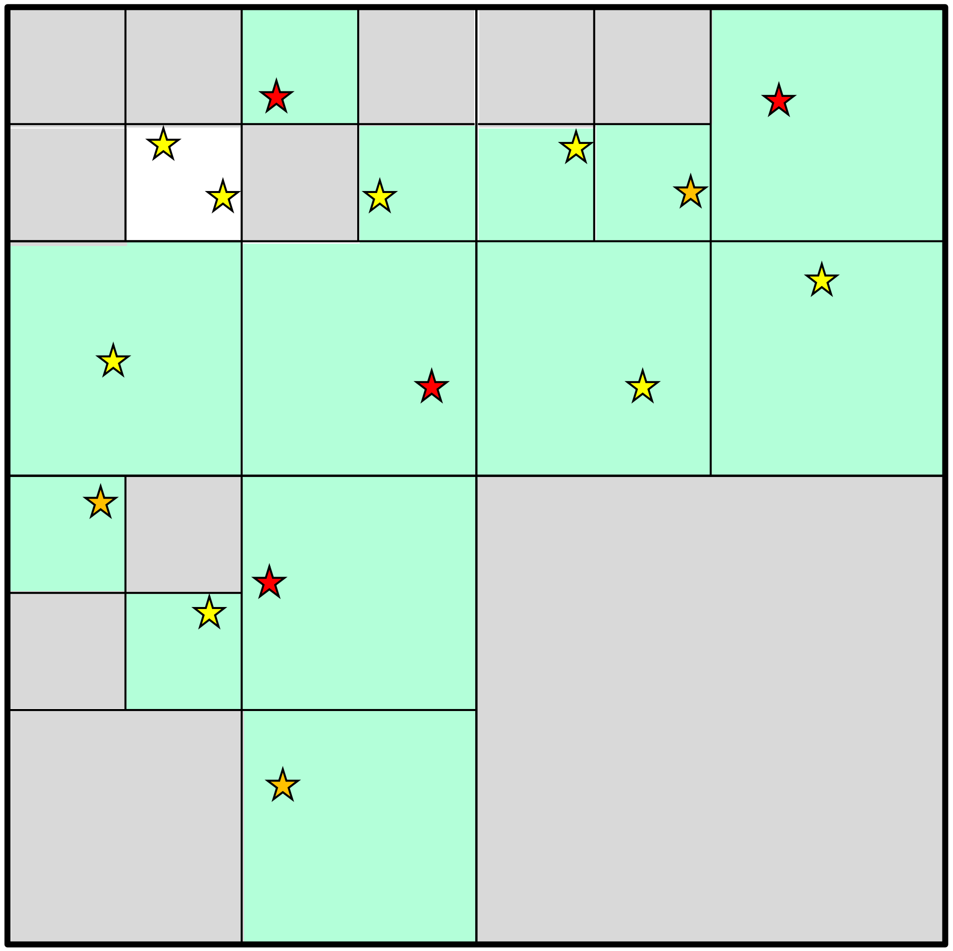
Barnes Hut Tree - Construction

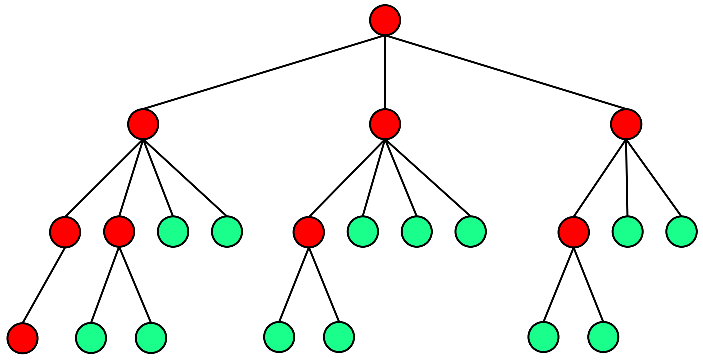




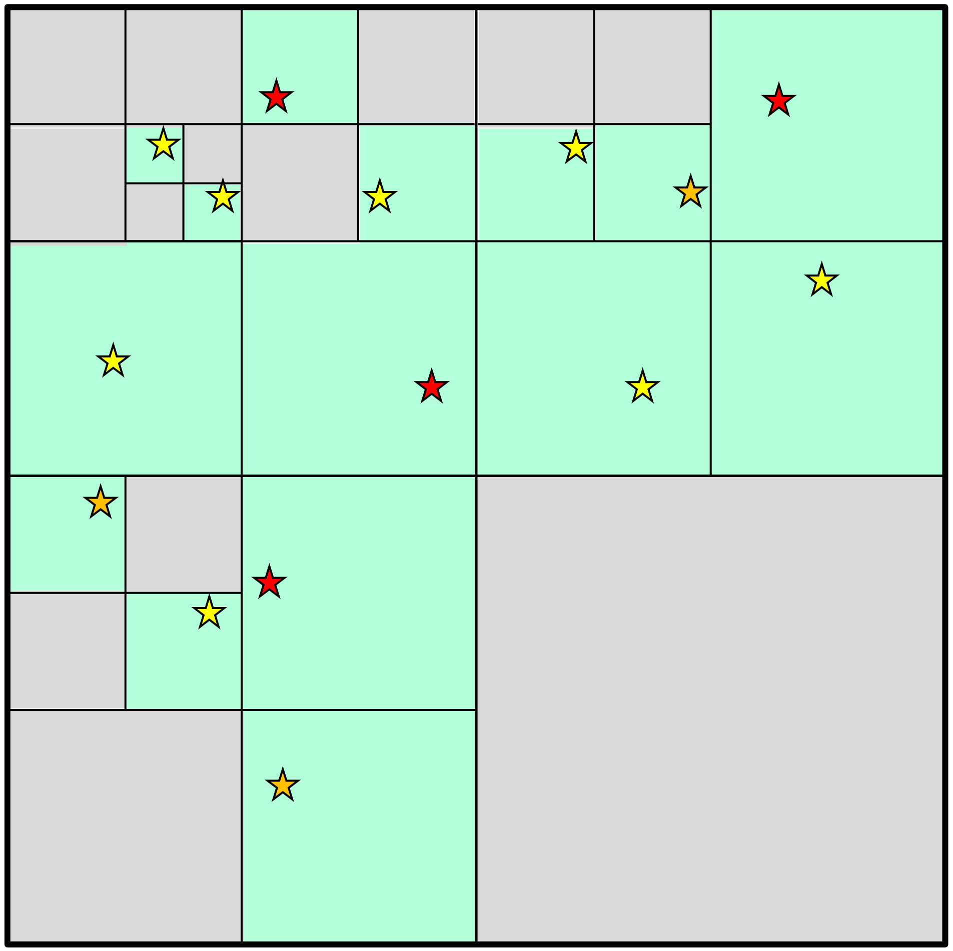
In the tree green is a single star, red is a node with children

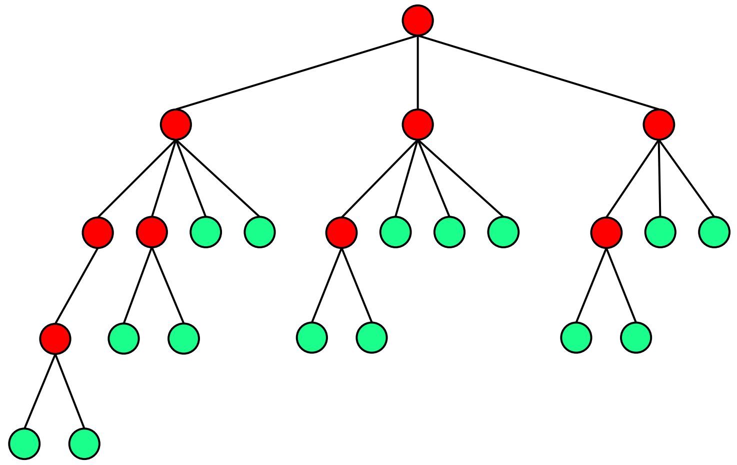
Barnes Hut Tree - Construction





Barnes Hut Tree - Construction





- Can have billions of stars with very deep BH tree
  - Want to construct quickly as likely need to update each timestep

- Processing each quadrant is embarrassingly parallel as long as the data structure can be updated concurrently

Barnes Hut Tree – Fork/Join

- Each quadrant of the space can be processed concurrently
  - Due to the irregular nature of the problem, fork/join is a natural way to construct the tree
  - Each time a quadrant needs to be subdivided, fork each child quadrant to a new UE.
  - A predictable amount of work per task and reasonably well balanced

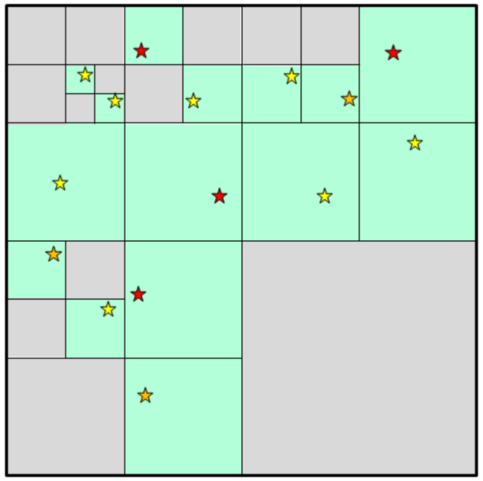
```
Node parentNode=...;
Starlist stars=...;

#pragma omp parallel
{
  #pragma omp single
  {
    evaluateNode(parentNode, stars);
  }
}
```

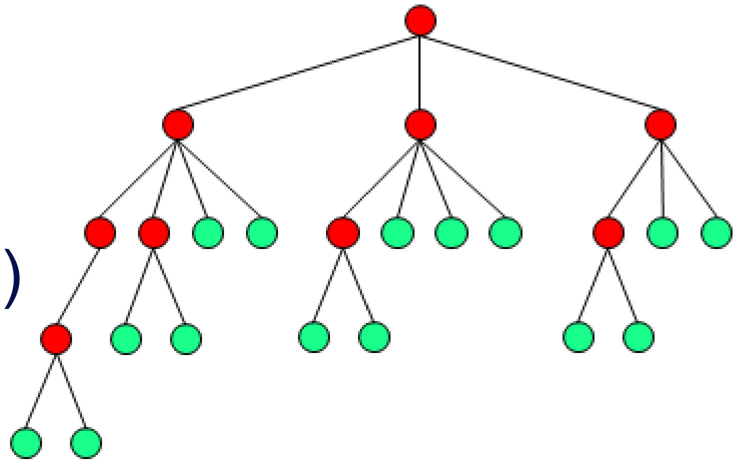
```
void evaluateNode(Node node, Starlist stars){
  int n=countStarsInNode(node,stars);
  DoSomeStuff();
  if (n > 1) {
    for (int i=0;i<4;i++){
      #pragma omp task
      {
        Quadrant newquad=divideQuad(quad,i);
        evaluateNode(newquad,stars);
      }
    }
    #pragma omp taskwait
  }
  doMoreStuff();
}
```

Barnes Hut – Evaluating Forces

Once the tree is constructed, for each particle, the tree is traversed, and the forces evaluated.



- Far enough away nodes do not need to be 'opened up' to their child nodes, and can be treated as single 'super' particles, with the total mass of the node, and its centre of mass as the location of the particle.



- Each particle (N) only has to have the force evaluated on a smaller number of 'particles' (typically log(N))
- Scales as N log(N) rather than N^2

Conclusions

- Fork/Join implementation strategy is suitable for irregular or dynamic control structures
  - Tasks are created (forked) and terminated (joined) dynamically
  - Might be that your tasks are all forked at the start and joined at the end, or might continually fork and join
- Direct mapping is a static load balancing scheme
  - Simplest but ties a task to a thread, so might result in oversubscription
- Indirect mapping is more of a dynamic scheme
  - More flexible, but not quite as flexible as the master/worker pattern
  - Prevents the runaway creation of threads
  - Cheaper to avoid repeated thread creation and destruction