UPC

Data distribution, synchronisation & work sharing

Nick Johnson
EPCC
Nick.Johnson@ed.ac.uk

- → data distribution
  - multi-dimensional data
- → synchronisation methods
  - blocking versus non-blocking
- → work sharing
  - examples: vector addition revisited, matrix-vector multiplication

Brief recap…

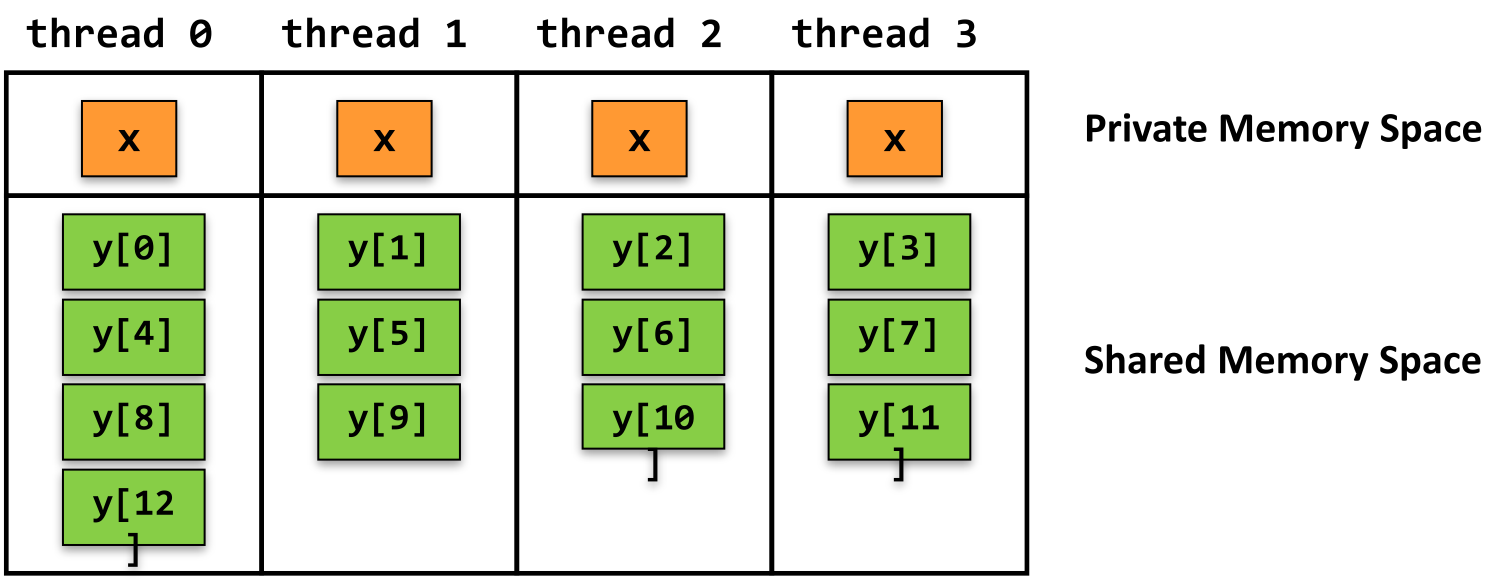
- ✓ private and shared data, logically partitioned memory space
- ✓ data objects have affinity to one thread exactly
- ✓ work sharing through `upc_forall`
- ✓ distribution of shared data
- ✓ storage duration of shared data
- ✓ synchronisation

focus of today's lecture

Data Distribution

Cyclic distribution is the default

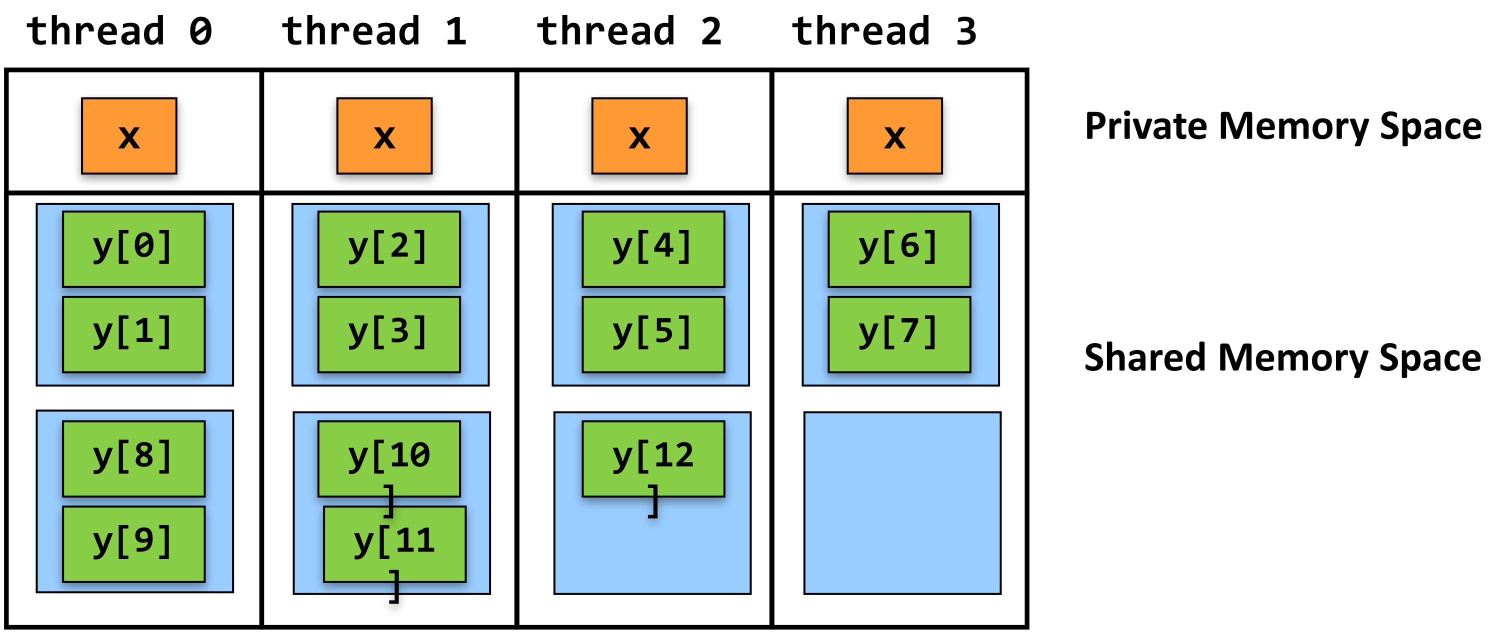
```
int x;
shared int y[13];
```



Data Distribution (2)

If number of elements is not an exact multiple of the thread count, threads can end up having uneven numbers of elements:

```
int x;
shared[2] int y[13];
```



Blocking factor

should be used if default distribution is not suitable

- more on the meaning of “suitable” later on…

➔ *four* different cases

`shared [4]` → defines a block size of 4 elements

`shared [0]` → all elements are given affinity to `thread 0`

`shared [*]` → when possible, data is distributed in contiguous blocks

`shared []` → equivalent to `shared [0]`

Multi-dimensional data

UPC can distribute data using ***block cyclic*** distributions

Distributions represent a ***top-down*** approach

- shared objects can be distributed into segments using the blocking factor

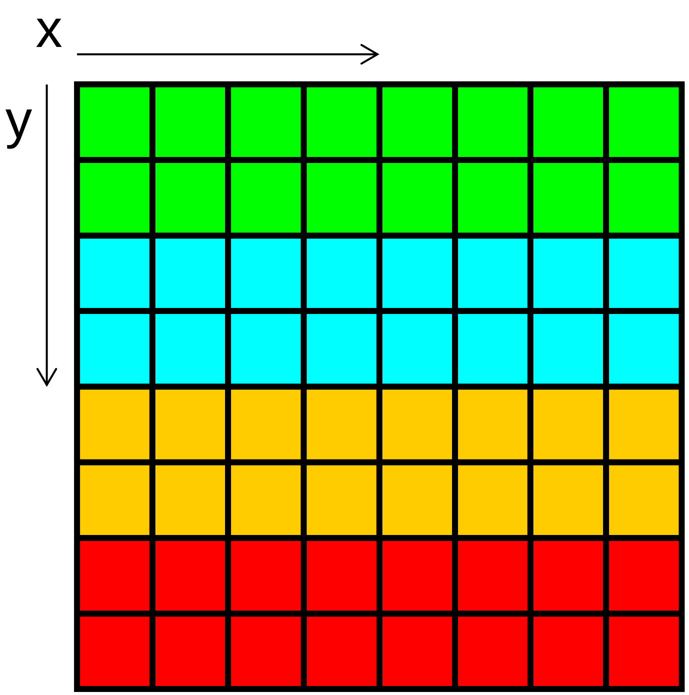
→ conceptually opposite of CAF, where shared objects are “created” by merging the pieces from every image in a bottom-up approach

2D array decomposition

distribution using the * layout qualifier and empty brackets

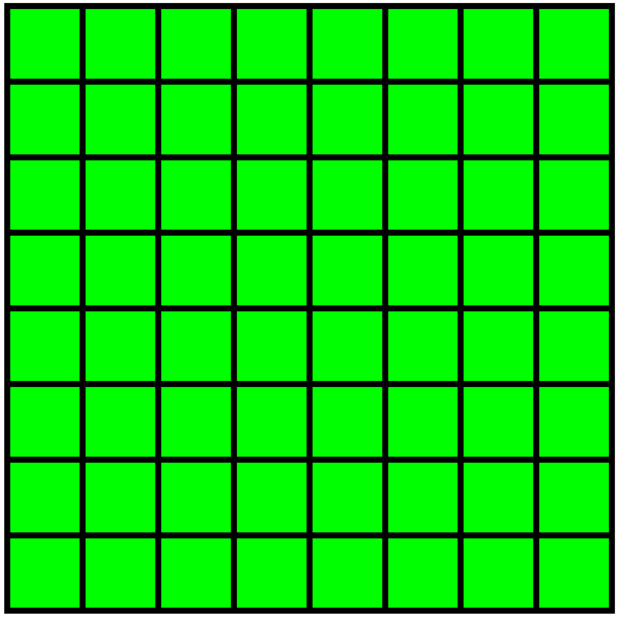
- Block distribution

```
shared[*] int y[8][8];
```



- Entire array on master

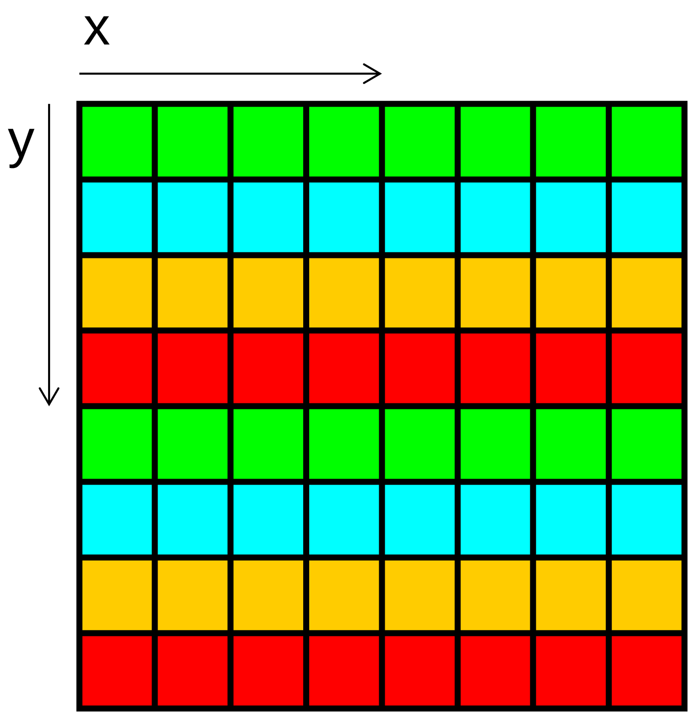
```
shared[]  int y[8][8]; or
shared[0] int y[8][8];
```



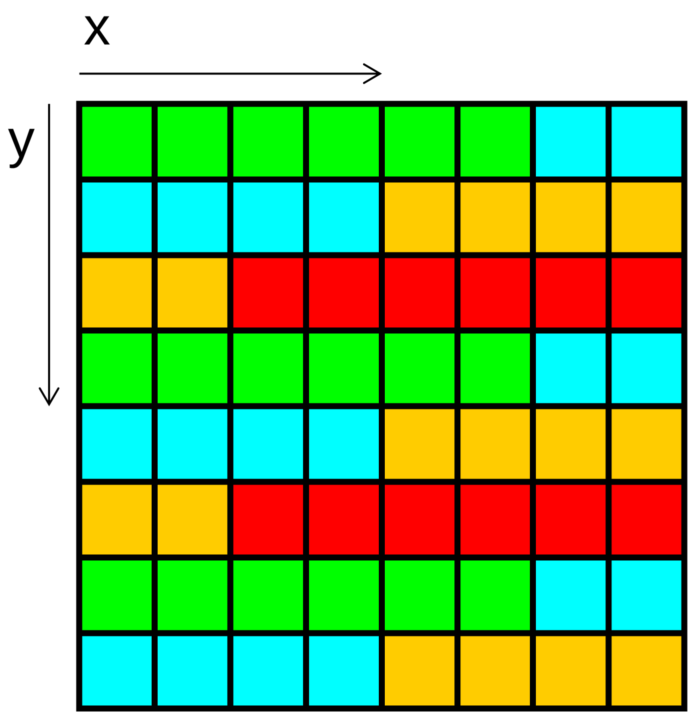
2D array decomposition

Distribution using different blocking factors

```
shared[8] int y[8][8];
```



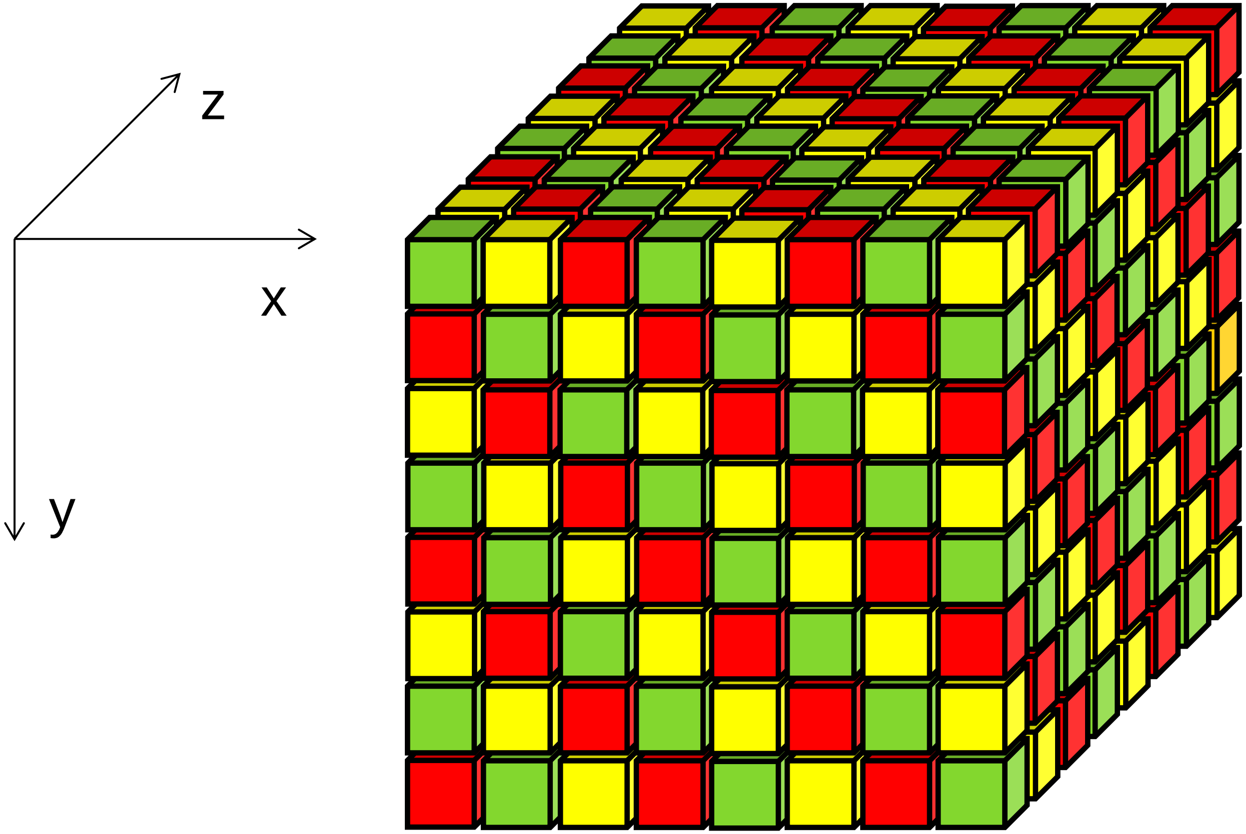

```
shared[6] int y[8][8];
```



N.B. array layout convention is arr[y][x]!

Multi-dimensional data – Case 1

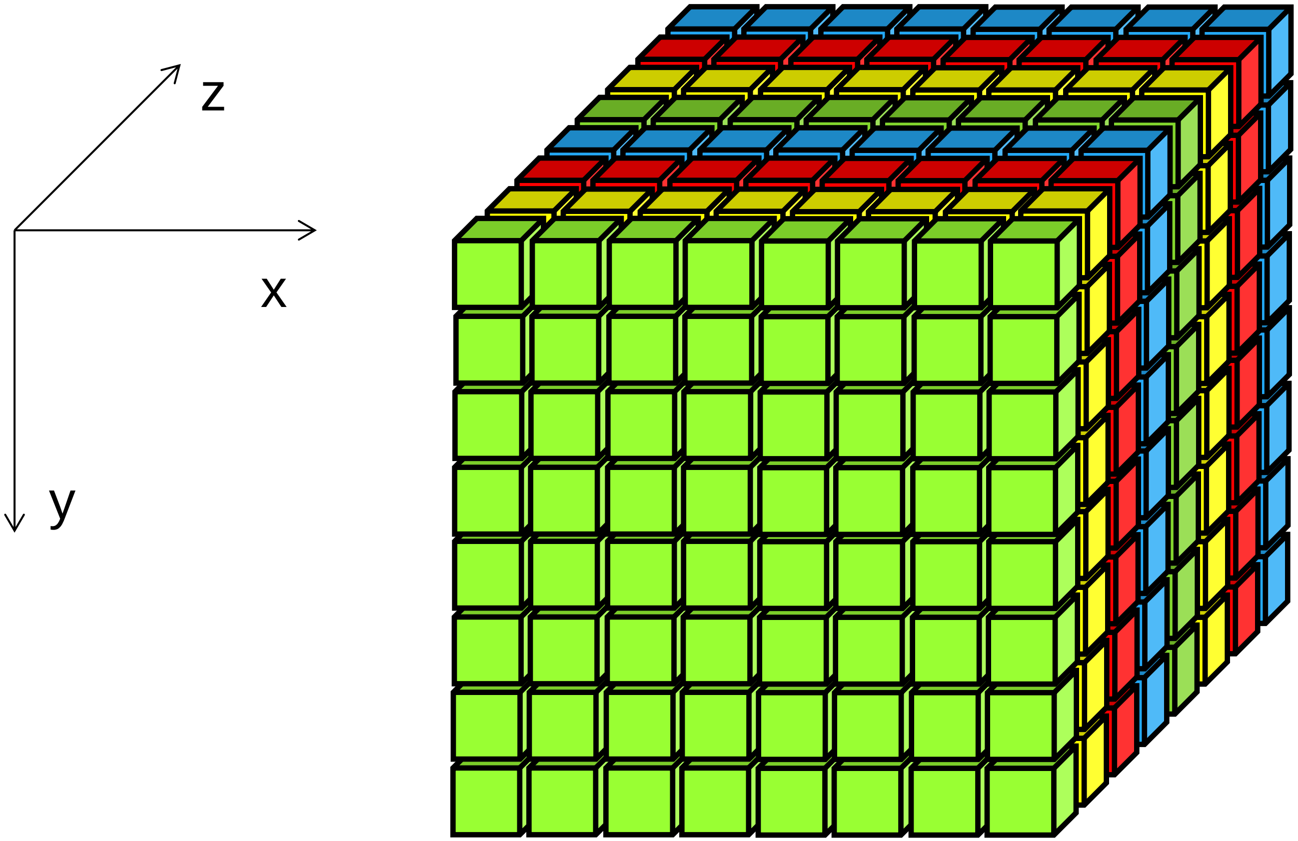
shared double grid[8][8][8] with **THREADS == 3**



N.B. array layout convention is arr[x][y][z]!

Multi-dimensional data – Case 1

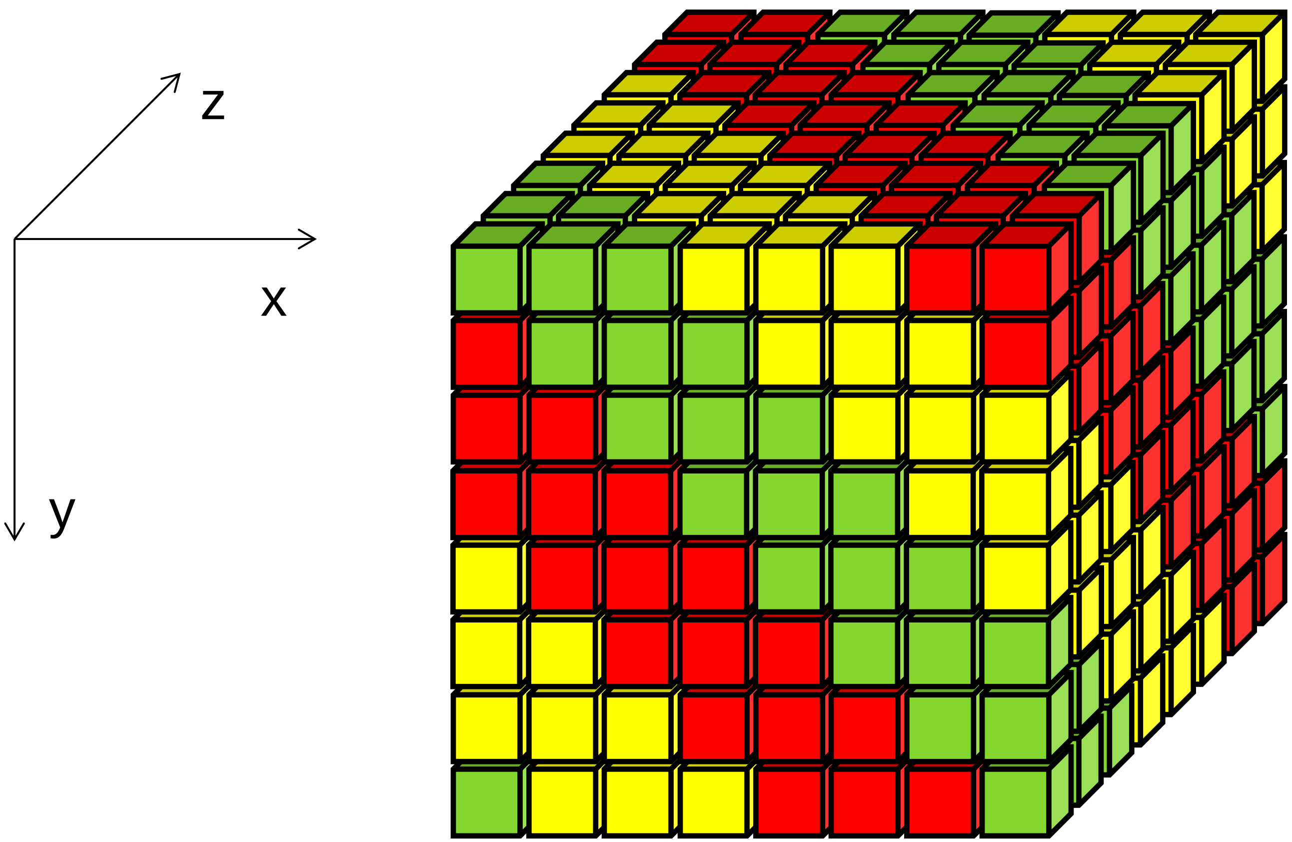
shared double grid[8][8][8] with **THREADS == 4**



N.B. array layout convention is arr[x][y][z]!

Multi-dimensional data – Case 2

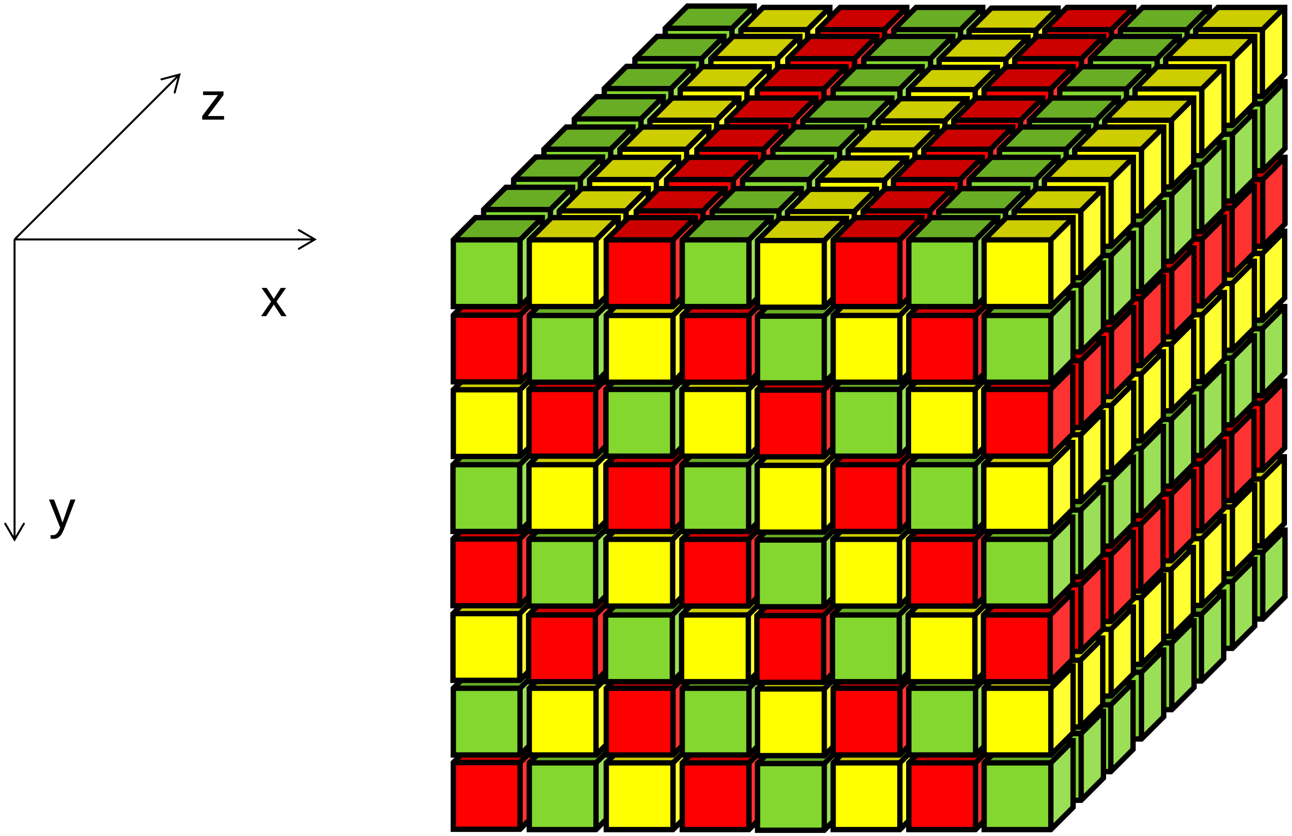
`shared [3] double grid[8][8][8]` with `THREADS == 3`



Multi-dimensional data – Case 2

blocking factor depending on dimensions → pencil distribution

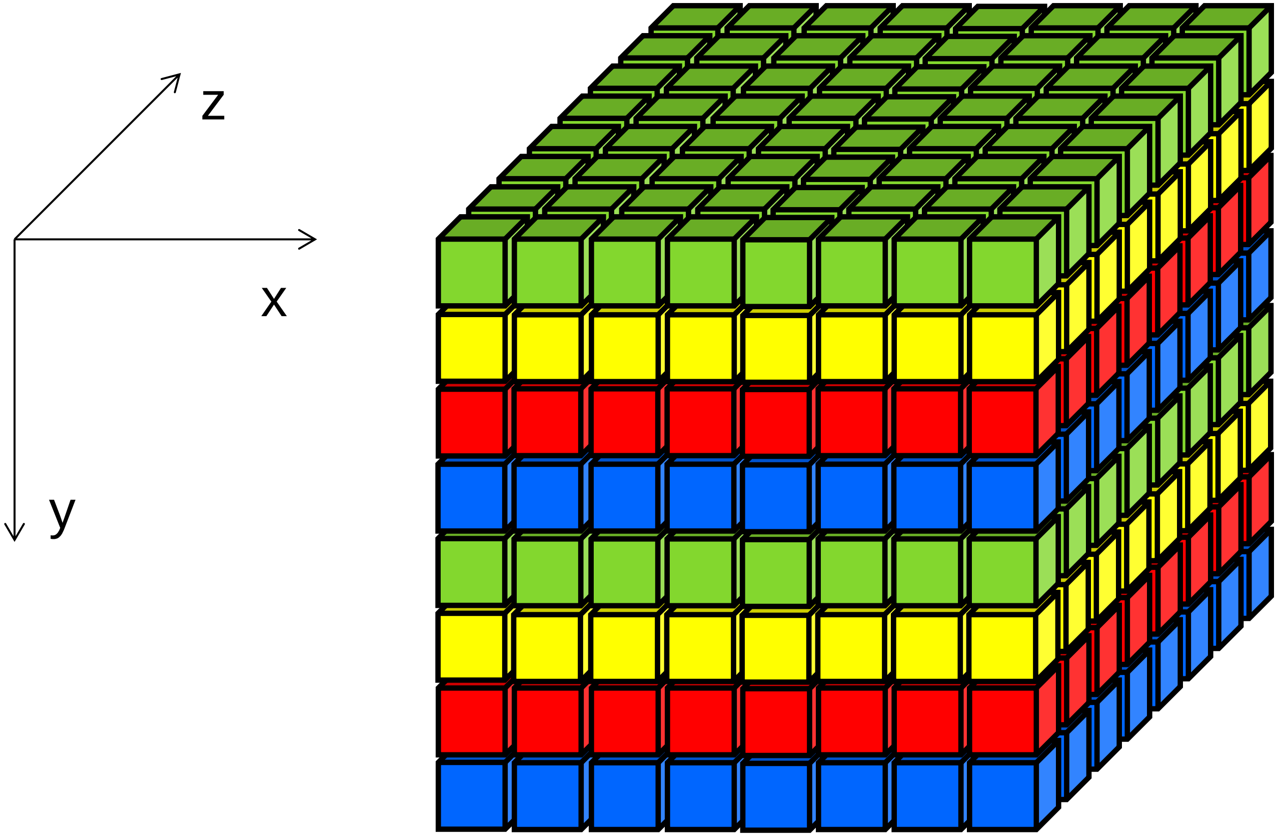
shared [8] double grid[8][8][8] with **THREADS == 3**



Multi-dimensional data – Case 2

combining thread count ***and*** blocking factor → slab distribution

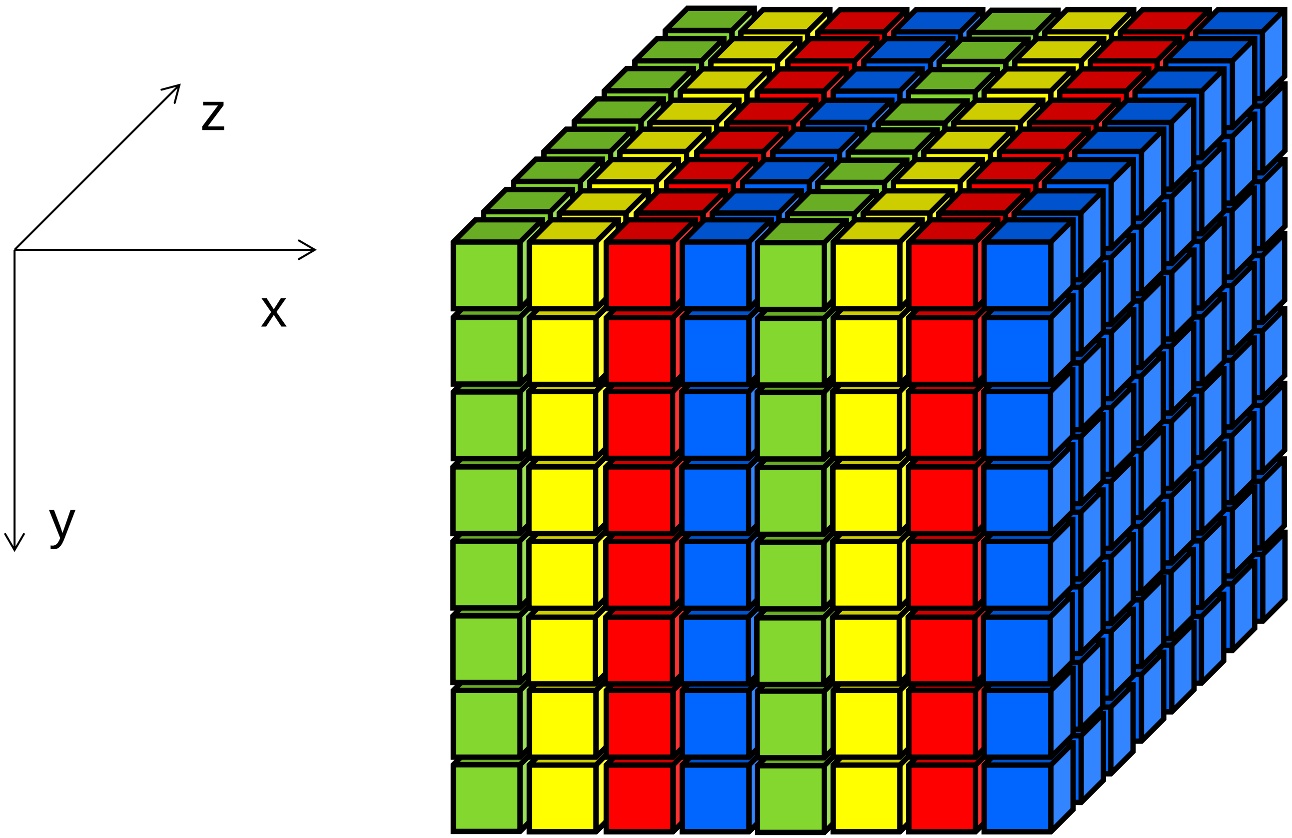
`shared [8] double grid[8][8][8]` with `THREADS == 4`



Multi-dimensional data – Case 3

slabs are contiguous in memory → blocking factor product of 2 dimensions

`shared [8*8] double grid[8][8][8]` with `THREADS == 4`



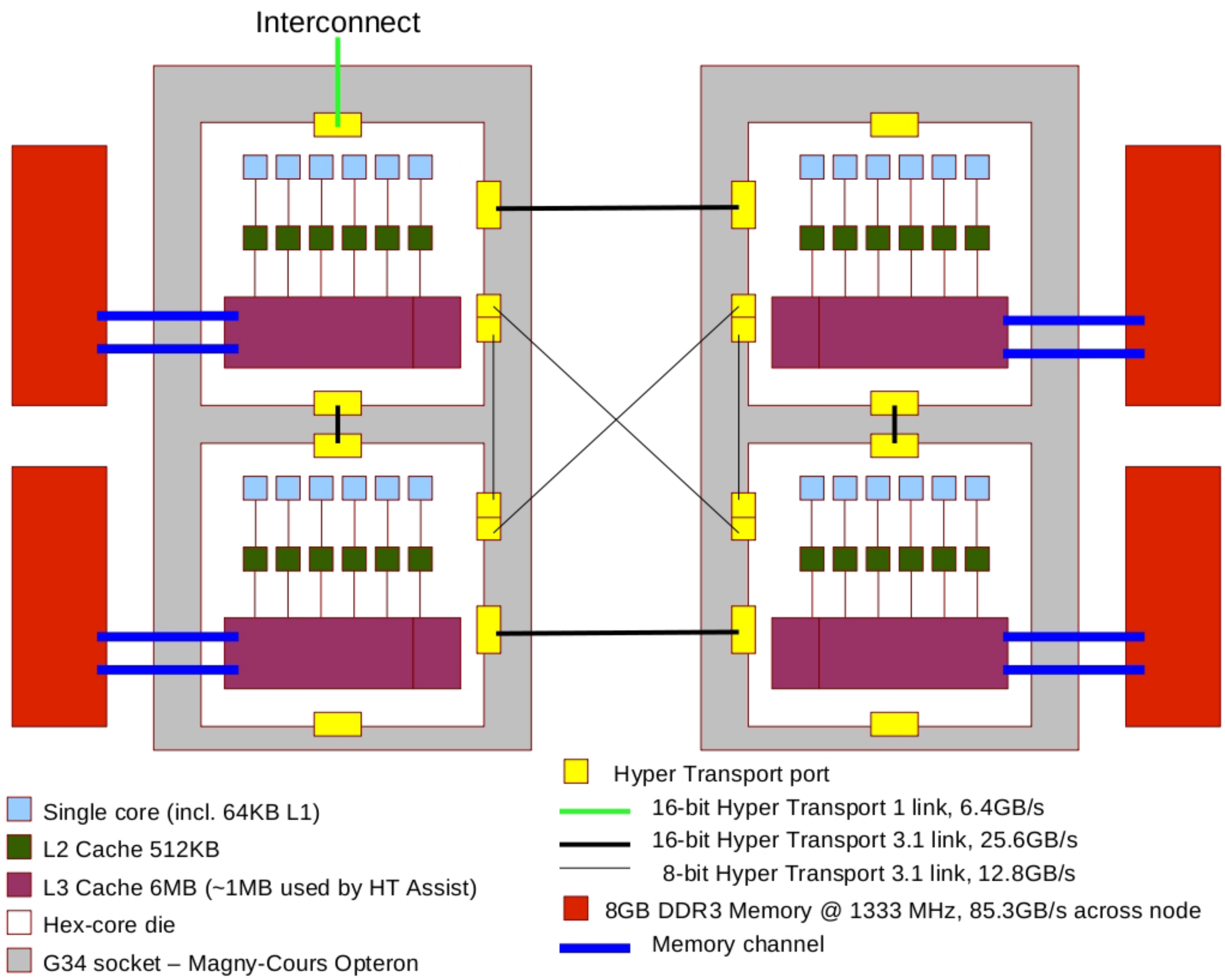
Why is the distribution important?

it is all about performance and minimising the cost of reading and writing data…

accessing shared data which resides in a physically remote location is ***more expensive*** than accessing shared data which has affinity with the thread!

optimise layout of data by using knowledge of problem size and, if possible, the number of threads

Why is the distribution important? (2)



Static vs. dynamic compilation (1)

number of UPC threads can be specified either at *compile time* *(static)* or at *runtime (dynamic)*

- GCCUPC Compiler: `-fupc-threads-N` specifies the number of threads at compile time as **N**

Advantages

- dynamic: program can be executed using any number of threads
- static: easier to distribute data based on THREADS

Disadvantages

- dynamic: not always possible to achieve best possible distribution
- static: program needs to be executed with number of threads specified at compile time

Static vs. dynamic compilation (2)

"An array declaration is illegal if THREADS is specified at runtime and the number of elements to allocate at each thread depends on THREADS."

```
shared int x[4*THREADS];
shared[] int x[8];
```

legal for static and dynamic environments

```
shared int x[8];
shared[] int x[THREADS];
shared int x[10+THREADS];
```

illegal for dynamic environment

Static vs. dynamic compilation (3)

static compilation can often give *better performance*, as data distribution is much easier to control

dynamic compilation provides *greater flexibility*, however optimal data distribution may not always be achievable

→ tradeoff between performance and convenience

Synchronisation

needed to ensure all threads reach same point in execution flow

- memory and data consistency
- two types of synchronisation: blocking and non-blocking

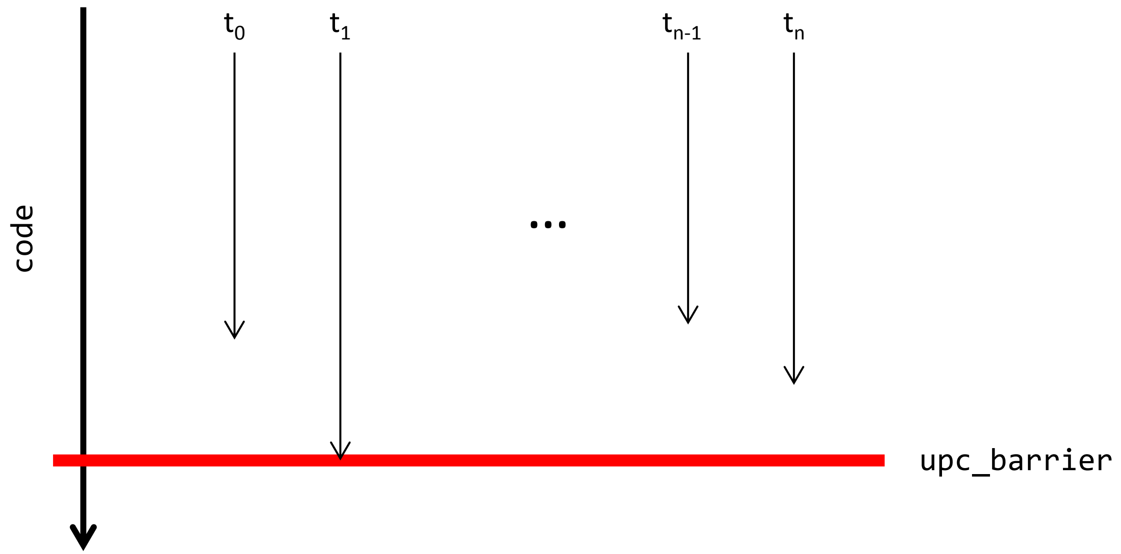
blocking synchronisation makes all threads wait at a barrier until the last thread has reached that barrier before allowing execution to continue

non-blocking synchronisation allows for some local computations to be executed while waiting for other threads

Barrier

`upc_barrier exp`$_{opt}$

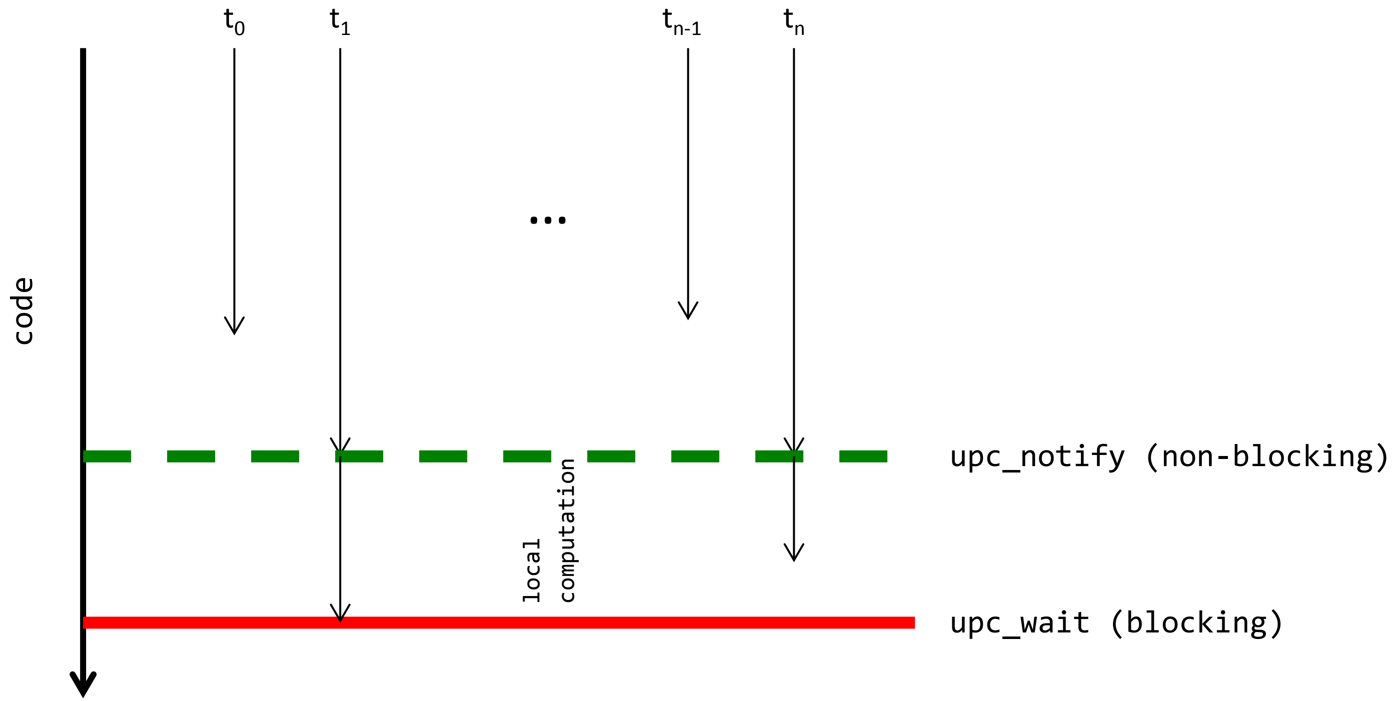
1. all threads execute the code that requires synchronisation
2. once finished they wait at the barrier
3. when the last thread reaches the barrier, all threads are released to continue execution



Split-phase barrier

`upc_notify exp`$_{opt}$ and `upc_wait exp`$_{opt}$

1. thread finishes work that requires synchronisation → upc_notify to inform others
2. thread performs local computations → once finished, wait
3. when all threads have execute upc_notify, thread waiting at barrier can continue



Debugging barriers

the optional value **exp** can be used to check that all threads have reached the same barrier

if a thread executes a barrier with different **exp** tag than the other threads, the application reports an expression mismatch and aborts

→ very useful for making sure that all threads are on the intended execution path

Work sharing revisited

4^{th} parameter in upc_forall loop represents *affinity*

if MYTHREAD executes an iteration

affinity is an integer expression

→ `affinity % THREADS == MYTHREAD`

affinity is a pointer-to-shared

→ object pointed to has affinity to `MYTHREAD`

→ **`upc_threadof(affinity)`**

Example: vector addition (1/3)

three vectors are distributed in cyclic fashion with the default blocking factor of 1

the modulo function identifies the local elements per thread

- if distribution changes, this code will *fail to identify the local elements*
- it will still produce the correct result!

```
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
   int i;

   for(i=0; i<N; i++)
       if (MYTHREAD == i%THREADS)
           v1plusv2[i] = v1[i] + v2[i];
}
```

Example: Vector addition (2/3)

alternative implementation:

iterate in steps of THREADS and eliminate the need for % operation

→ this implementation is again distribution specific

```
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
   int i;

   for(i=MYTHREAD; i<N; i+=THREADS)
           v1plusv2[i] = v1[i] + v2[i];
}
```

Example: Vector addition (3/3)

Advantage of affinity parameter: even if the distribution changes, **`upc_forall`** will behave correctly *and* identify local elements

```
#include <upc.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main() {
   int i;

   upc_forall(i=0; i<N; i++; i)
           v1plusv2[i] = v1[i] + v2[i];
}
```

affinity parameter is integer expression

Work sharing and distribution example

matrix-vector multiplication

→ perform as much computation on local data as possible

→ work distribution is based on the elements of vector **c**

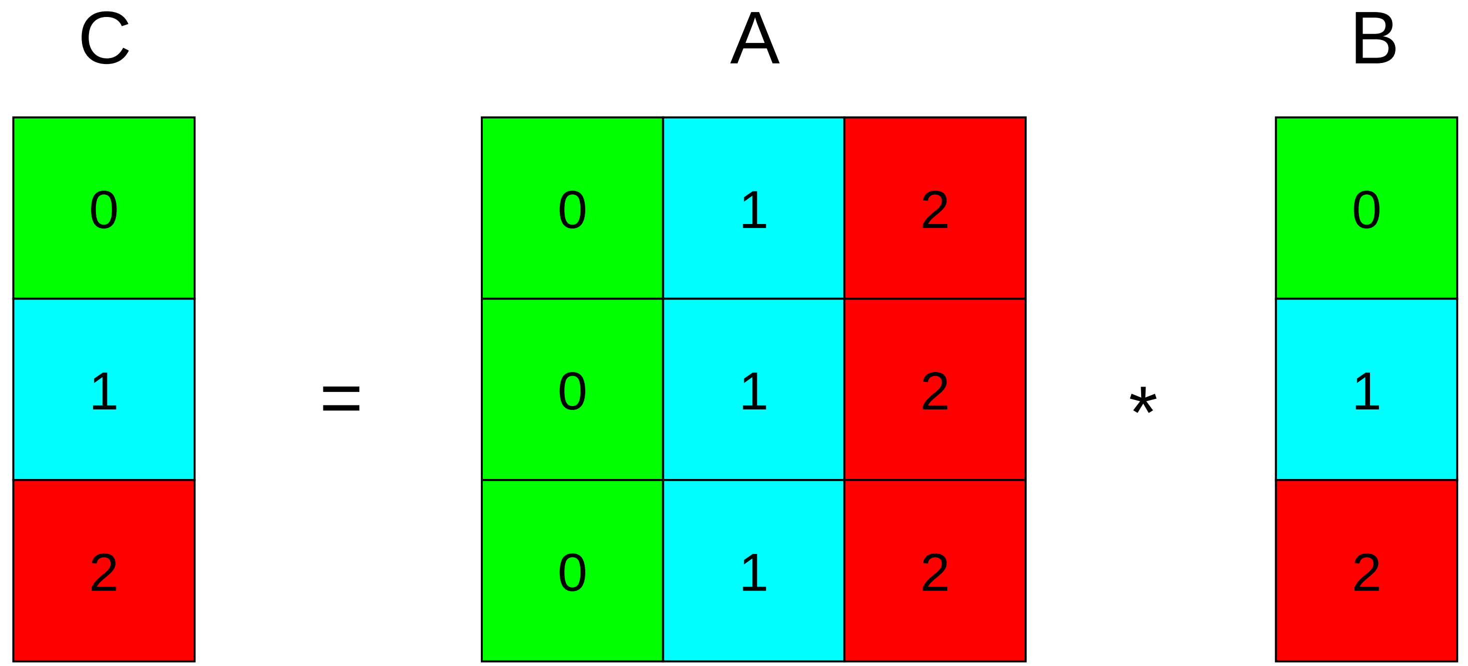
```
#include <upc.h>
shared int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void)
{
   int i, j;

   upc_forall( i = 0 ; i < THREADS ; i++; &c[i]) {
       c[i] = 0;
       for ( j= 0 ; j < THREADS ; j++)
           c[i] += a[i][j]*b[j];
   }
}
```

affinity parameter is pointer to shared array

Data distribution



number of remote operations:

$C_0 = A_{0,0}B_0 + A_{0,1}B_1 + A_{0,2}B_2$ → 4

$C_1 = A_{1,0}B_0 + A_{1,1}B_1 + A_{1,2}B_2$ → 4

$C_2 = A_{2,0}B_0 + A_{2,1}B_1 + A_{2,2}B_2$ → 4

We can do better

distribute matrix a in blocks of size **THREADS**

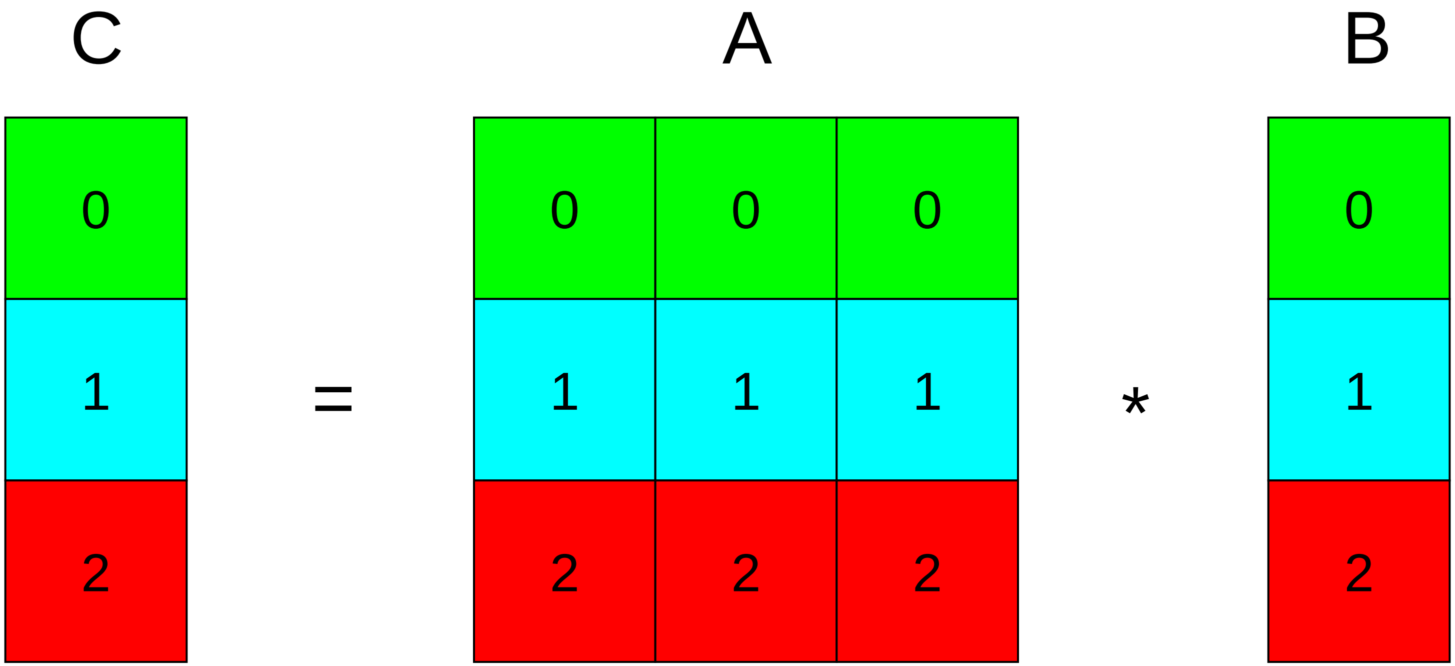
→ each row will be placed locally onto each thread

```
#include <upc.h>
shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void)
{
    int i, j;

    upc_forall( i = 0 ; i < THREADS ; i++;&c[i]) {
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```

Improved data distribution



number of remote operations:

$C_0 = A_{0,0}B_0 + A_{0,1}B_1 + A_{0,2}B_2$ → 2

$C_1 = A_{1,0}B_0 + A_{1,1}B_1 + A_{1,2}B_2$ → 2

$C_2 = A_{2,0}B_0 + A_{2,1}B_1 + A_{2,2}B_2$ → 2

Summary

correct data distribution is important for performance

- keep the number of remote reads/writes as low as possible

UPC gives programmers control over data layout and work sharing

- → it is important to be aware of performance implications
- → aim to keep work sharing loops independent of data distribution