# UPC

## Introduction & Basics

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

Objectives of the coming three lectures:

- understand the basic principles of UPC
- motivation behind PGAS
- learn about data distribution, synchronisation
- advanced features (dynamic memory allocation, collectives)

→ Practicals will try and emphasise the most important aspects of UPC

Unified Parallel C

Parallel extension to ISO C 99, adding

- explicit parallelism
- global shared address space
- synchronisation

Both commercial and open source compilers available

- Cray, IBM, SGI, HP
- GWU, LBNL, GCCUPC

# UPC and the World of PGAS

UPC != PGAS

- PGAS is a programming model
- UPC is *one* implementation of this model

Many other implementations

- Language extension: Coarray Fortran
- New languages: Chapel, X10, Fortress, Titanium
- PGAS-like libraries: OpenSHMEM, Global Arrays

All implementations are different, but follow the same model!

# UPC threads

UPC uses threads that operate independently in a SPMD fashion

→ threads execute the same UPC program

Identifiers that return information about the program environment:

**`THREADS`:** holds total number of threads

**`MYTHREAD`:** stores thread index

→ index runs from **`0`** to **`THREADS-1`**

# UPC threads

```
#include <upc.h>
#include <stdio.h>

void main() {
   printf("Thread %d of %d says: Hello!", MYTHREAD, THREADS);
}
```

# Private vs. shared memory space

Concept of two memory spaces: **private** and **shared**

objects declared in **private** memory space are only accessible by a single thread

objects declared in **shared** memory space are accessible by all threads

➔ shared memory space is used to communicate information between threads

# Private vs. shared data

**private** variables declared as normal C variable

- multiple instances of variable will exist

```
int x; // private variable
```

**shared** variables declared with **shared** qualifier

- only allocated once, in shared memory space
- accessible by all threads

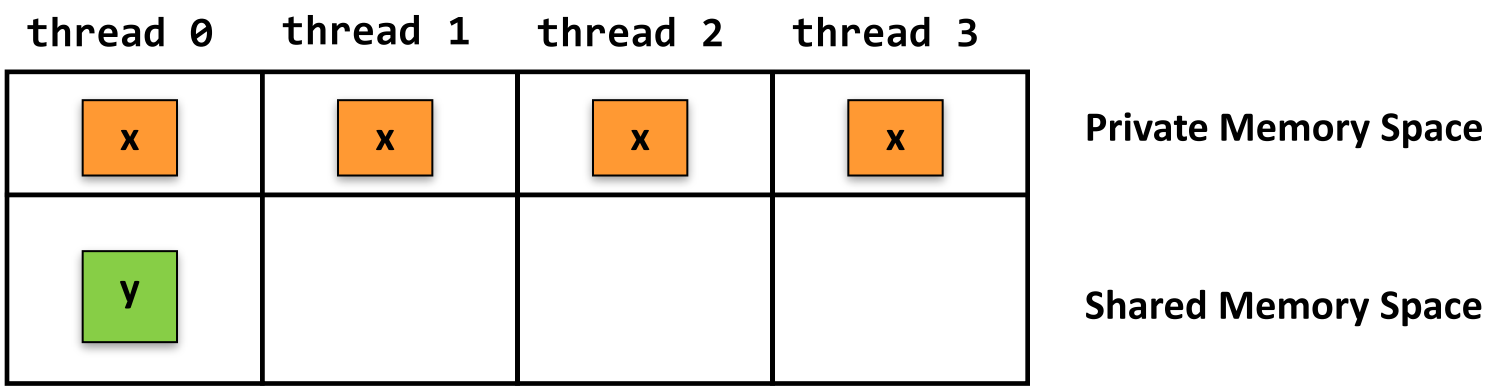
```
shared int y; // shared variable
```

# UPC data locality

If shared variable is scalar, space only allocated on `thread 0`

```
int x;

shared int y;
```



# Affinity

all threads can directly access shared data, even if it resides in a remote location

UPC creates logical partitioning of the shared memory space

→ objects have *affinity* to one thread

→ shared scalars always have affinity to `thread 0`

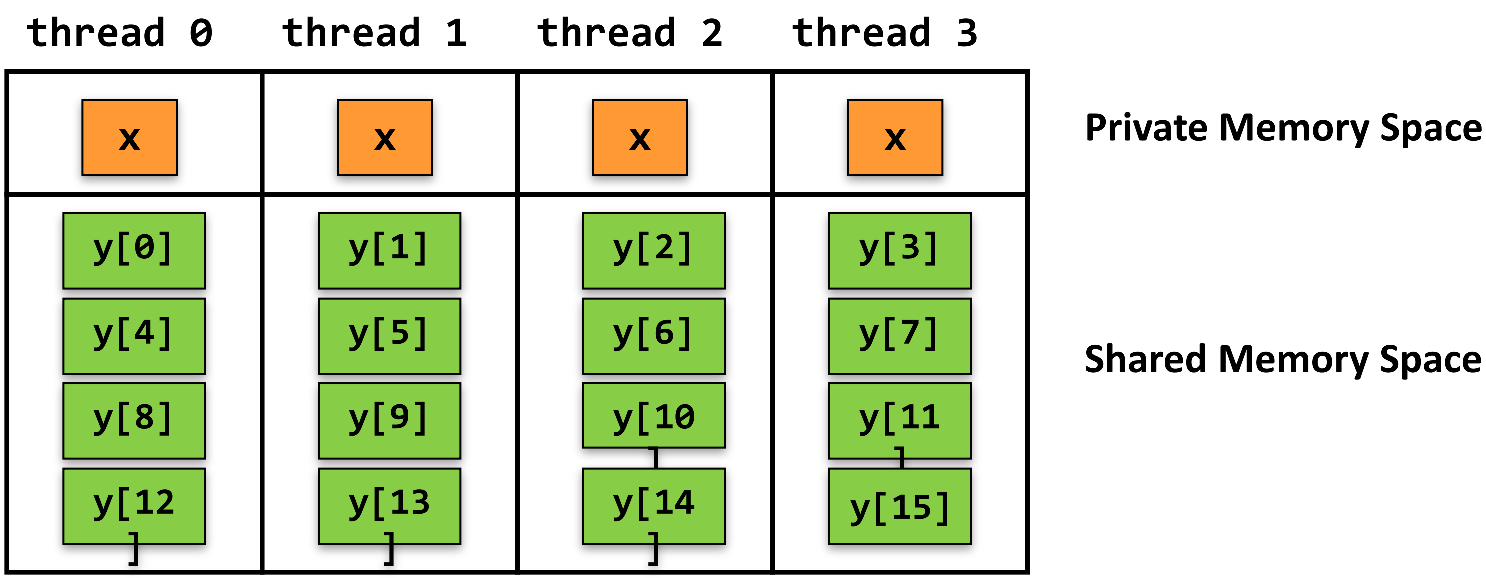
better performance if a thread access data to which it has affinity

→ always keep data locality and affinity in mind

# Shared array distribution

If a shared variable is an array, space allocated across shared memory space in a *cyclic* fashion by default

```
int x;
shared int y[16];
```

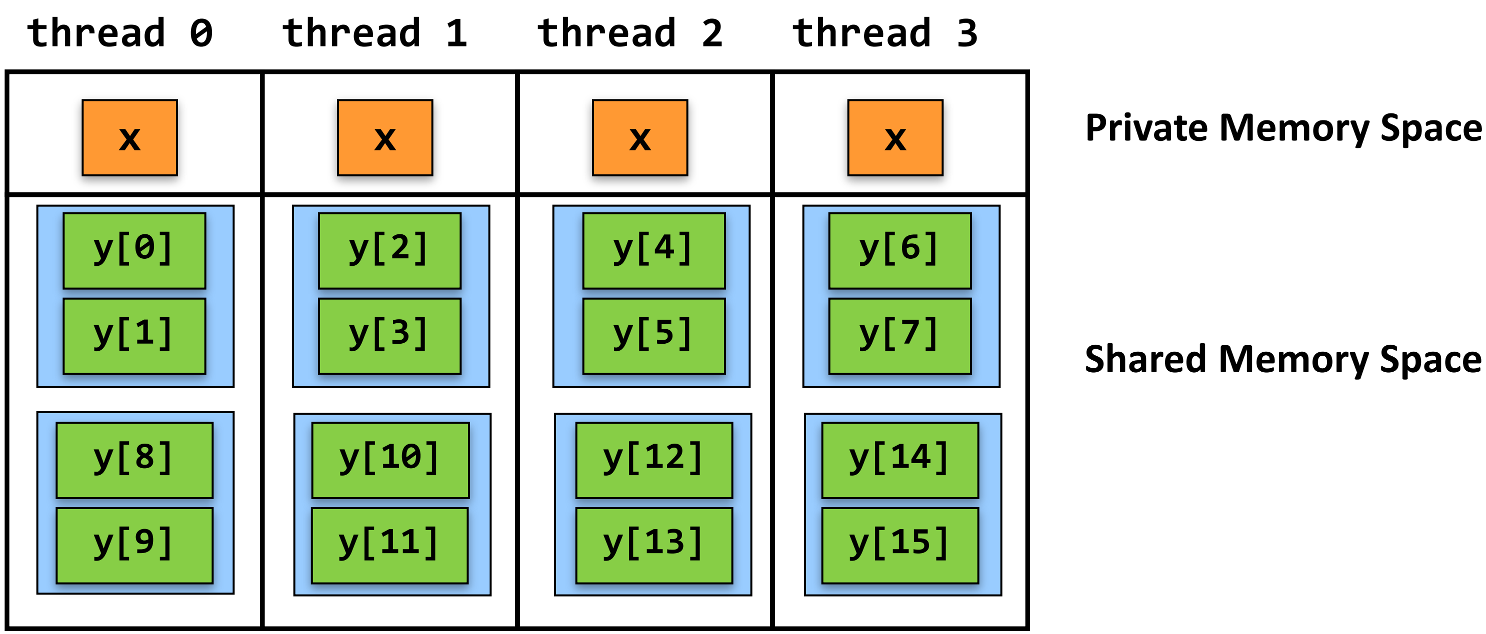


# Shared array distribution (2)

Change data layout by adding a “blocking factor” to shared arrays

```
shared[blocksize] type array[n]
```

```
int x;
shared[2] int y[16];
```



# Work sharing

Shared data means shared workload!

If shared data is distributed between threads, threads can distribute work on this data between them

UPC has built-in mechanism for explicitly distributing and sharing work

# Work sharing: **upc_forall**

Statement for work distribution

- allows loop assignment of tasks to threads
- *parallel* `for` loop

4th parameter defines affinity to thread

- if "`affinity % THREADS`" matches MYTHREAD, execute iteration for that THREAD

```
upc_forall(expression; expression; expression; affinity)
```

*Condition*: iterations of **`upc_forall`** must be independent!

# Example: vector addition (1)

```
#define N 10 * THREADS
shared int vector1[N];
shared int vector2[N];
shared int sum[N];

void main() {
   int i;
   for(i=0; i<N; i++){
      sum[i] = vector1[i] + vector2[i];
   }
}
```

# Example: vector addition (2)

```
#define N 10 * THREADS
shared int vector1[N];
shared int vector2[N];
shared int sum[N];

void main() {
   int i;
   upc_forall(i=0; i<N; i++; i){
      sum[i] = vector1[i] + vector2[i];
   }
}
```

evaluated as i%THREADS

# Side effects of shared data

Holding data in a shared memory space has implications

1) the lifetime of shared data needs to extend beyond the scope it was defined in (unless this is program scope)

→ storage duration

2) the shared data needs to be keep up-to-date

→ synchronisation

# Storage duration of shared objects

Shared objects *cannot* have automatic storage duration

- any variable defined inside a function!

*Why?*

SPMD model means a shared variable may be accessed outside lifetime of the function!

*Conclusion*

shared variables must either

- have file scope;
- or be declared as `static` if defined inside a function.

# “Static” keyword

ensures shared objects are accessible throughout program execution

→ objects are not linked to the scope of a thread

→ objects will not simply “disappear” after a thread exists the scope in which the object was defined

# Example: maximum of an array

```
#define max(a,b) (((a)>(b)) ? (a) : (b))
shared int maximum[THREADS];
shared int globalMax = 0;
shared int a[THREADS*10];

void main(int argc, char **argv) {
        … // initialise array a

        upc_forall(int i=0; i<THREADS*10; i++; i){
                  maximum[MYTHREAD] = max(maximum[MYTHREAD], a[i]);
        }

        if (MYTHREAD == 0){
           for (int thread=0; thread<THREADS; thread++){
                    globalMax = max(globalMax,maximum[thread]);
           }
        }

…
}
```

Here: shared variables have file scope!

This code will not work!!

# Synchronisation

Ensure all threads reach same point in execution

- necessary for memory and data consistency

Barriers used for synchronisation

- blocking
- split-phase (non-blocking)

`upc_barrier` → blocking

`upc_notify, upc_wait` → non-blocking

# Example: maximum of an array

```
#define max(a,b) (((a)>(b)) ? (a) : (b))
shared int maximum[THREADS];
shared int globalMax = 0;
shared int a[THREADS*10];

void main(int argc, char **argv) {
        … // initialise array a
        upc_barrier;
        upc_forall(int i=0; i<THREADS*10; i++; i){
                maximum[MYTHREAD] = max(maximum[MYTHREAD], a[i]);
        }
        upc_barrier;
        if (MYTHREAD == 0){
           for (int thread=0; thread<THREADS; thread++){
                  globalMax = max(globalMax,maximum[thread]);
           }
        }
        upc_barrier;
…
}
```

Here: shared variables have file scope!

Ensure all threads found local maximum

Makes sure `globalMax` is found before being used

# References

- UPC Language Specification (Version 1.2):

  http://upc.gwu.edu/docs/upc_specs_1.2.pdf

- UPC homepage:

  http://upc.gwu.edu/

- GCCUPC compiler:
- http://www.gccupc.org