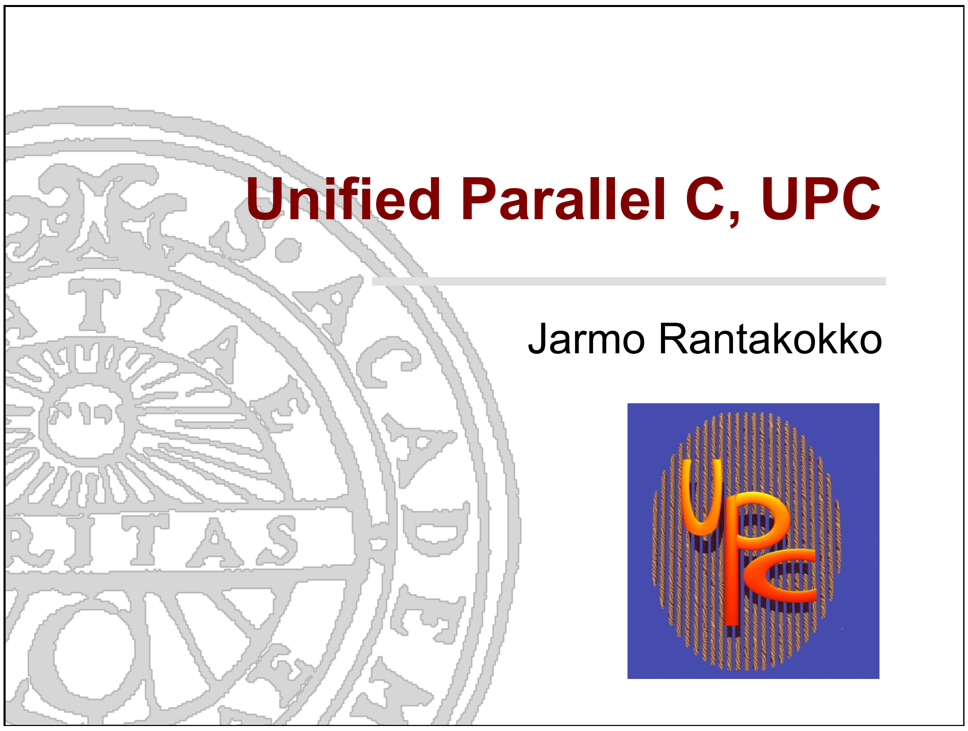
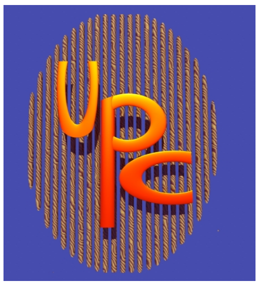
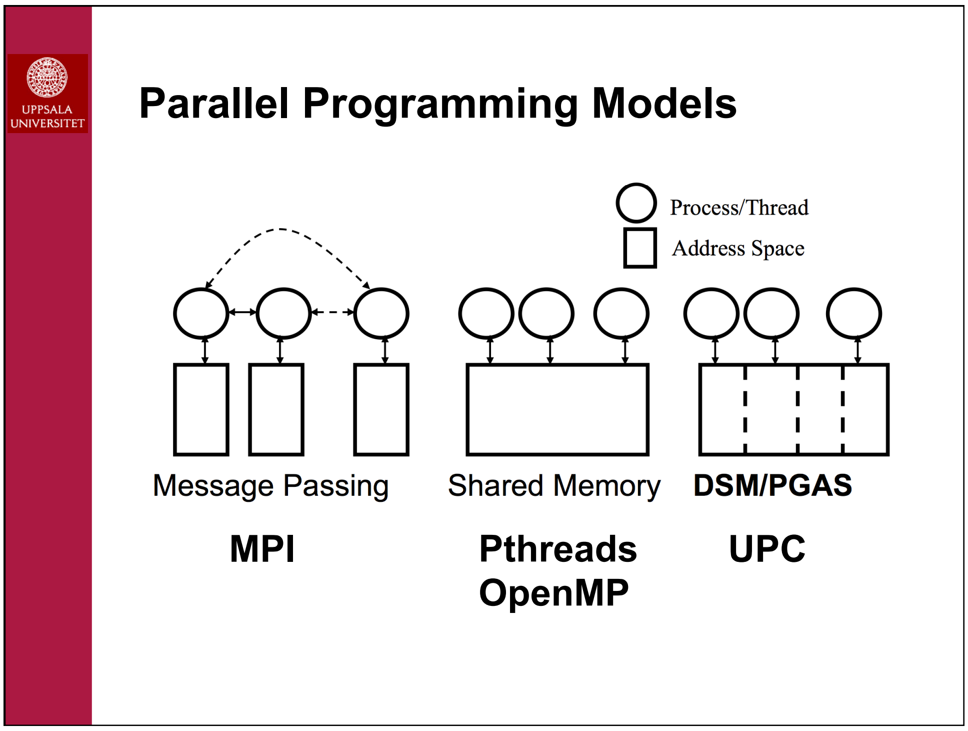


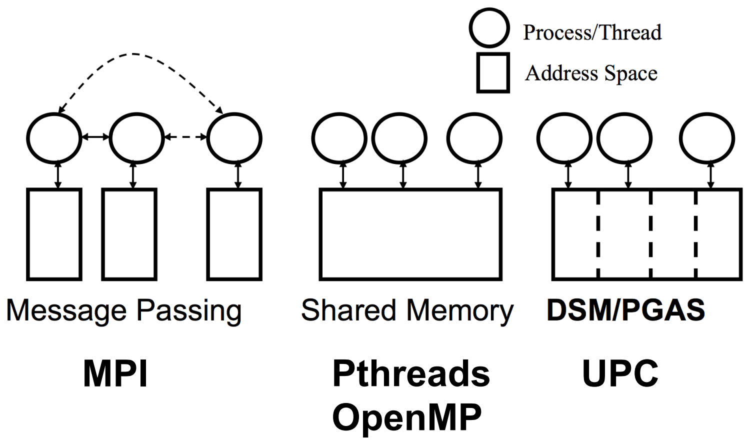
Unified Parallel C, UPC

Jarmo Rantakokko

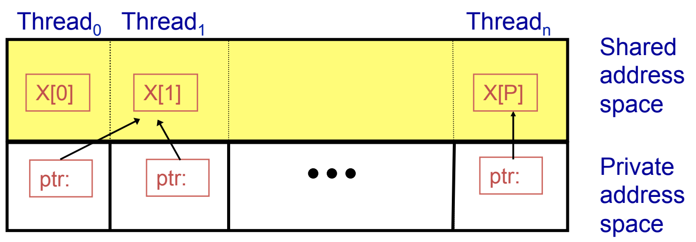




Parallel Programming Models



Partitioned Global Address Space, PGAS



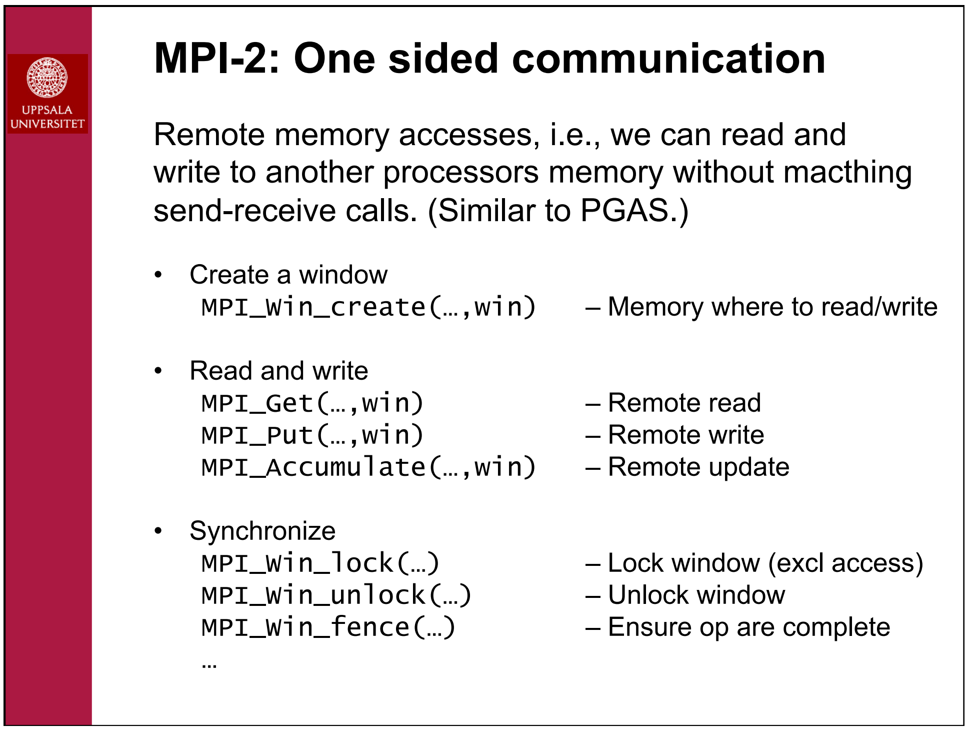
The languages share the global address space abstraction

- Shared memory is logically partitioned by processors
- Global arrays have fragments in multiple partitions
- Remote memory may stay remote: no automatic caching implied
- One-sided communication: reads/writes of shared variables
- Both individual and bulk memory copies

+ Simple as shared memory, helps in exploiting locality
- Can result in subtle bugs and race conditions

Implementations of PGAS

- **UPC, Unified Parallel C**
  v1.0 completed February of 2001
  v1.3 released November 2013
- **CAF, Co-Array Fortran**
  v1.0 completed 1998
  Now, included in Fortran 2008 standard
- **Titanium (Berkeley), X10 (IBM)**
  PGAS extensions to JAVA
- **Chapel**
  New language developed by Cray inc
  v0.94 released October 2013



MPI-2: One sided communication

Remote memory accesses, i.e., we can read and write to another processors memory without macthing send-receive calls. (Similar to PGAS.)

- Create a window

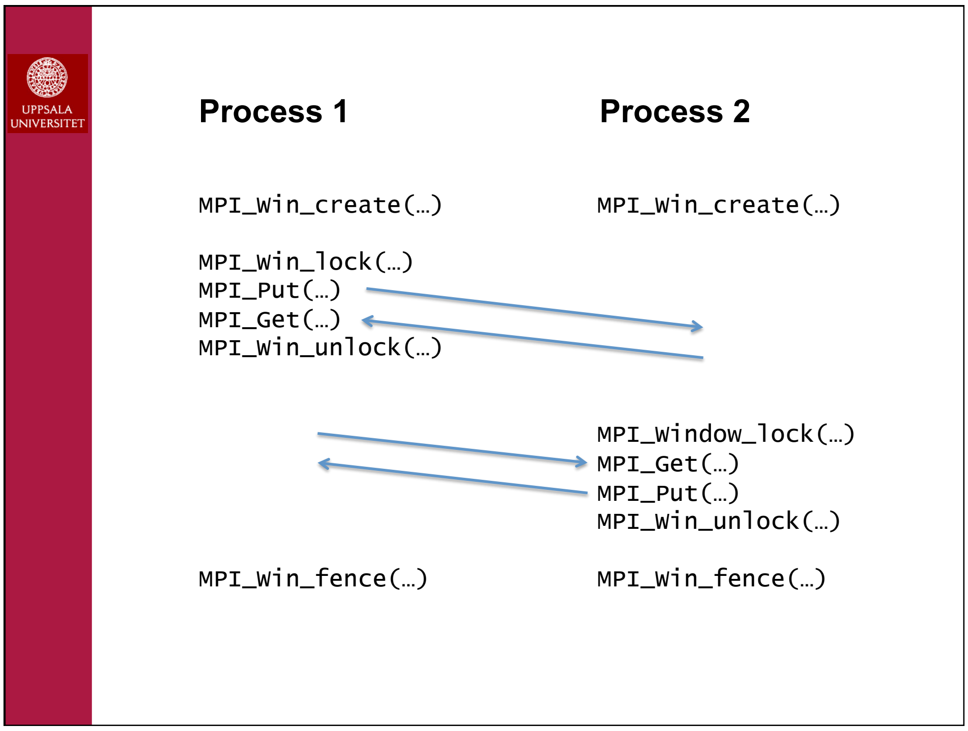
`MPI_Win_create(…,win)`	– Memory where to read/write

- Read and write

`MPI_Get(…,win)`	– Remote read
`MPI_Put(…,win)`	– Remote write
`MPI_Accumulate(…,win)`	– Remote update

- Synchronize

`MPI_Win_lock(…)`	– Lock window (excl access)
`MPI_Win_unlock(…)`	– Unlock window
`MPI_Win_fence(…)`	– Ensure op are complete
…	



```
Process 1                         Process 2

MPI_Win_create(…)                 MPI_Win_create(…)

MPI_Win_lock(…)
MPI_Put(…)       ------------->
MPI_Get(…)       <-------------
MPI_Win_unlock(…)

                 ------------->   MPI_Window_lock(…)
                                  MPI_Get(…)
                 <-------------   MPI_Put(…)
                                  MPI_Win_unlock(…)

MPI_Win_fence(…)                  MPI_Win_fence(…)
```

UPC Language at a glance

- A partitioned global address space language
- SPMD model with independent threads
- An explicit parallel extension of ANSI C
  - `shared` for shared global data
  - `[partition]` for data layout of arrays
  - `forall` for parallel execution
  - Synchronization with `barrier,fence,lock`
  - Collective operations, `broadcast, scatter, gather, reduce, prefix`
  - Shared dynamic memory allocation, `upc_all_alloc, upc_global_alloc`
  - Get owner/affinity of data, `upc_threadof`
  - Different pointer types to shared data

Hello World in UPC

- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.

```
#include <upc_relaxed.h>  /* needed for UPC */
#include <stdio.h>
main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

Compile and run:

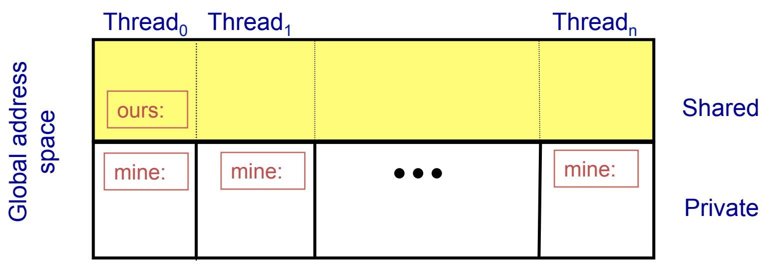
```
> upcc -pthreads -o hello helloworld.c
> upcrun -n 4 hello
```

Private vs shared variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

```
shared int ours; // Shared on thread 0
int mine;        // Private on all threads
```

- Shared variables may not have dynamic lifetime: may not occur in a in a function definition, except as static.



Private vs shared variables in UPC

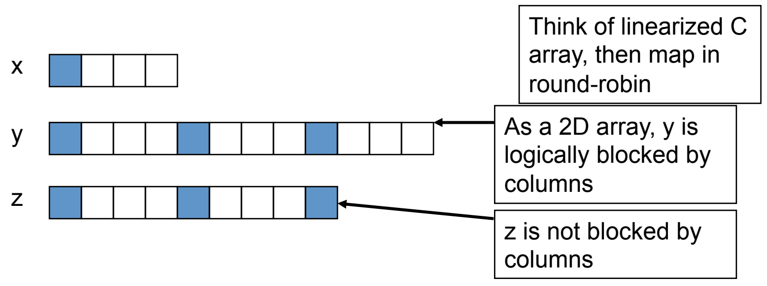
```
#include <upc_relaxed.h>
#include <stdio.h>

shared int ours; //Shared data
main() {
  int mine;      // Private data
  mine=MYTHREAD;
  if (MYTHREAD==THREADS-1) ours=-1;
  upc_barrier;
  printf(”Mine %d and ours %d\n”,
         mine,ours);
}
```

Shared arrays in UPC

- Shared scalars always live in thread 0
- Shared arrays are spread cyclicly over the threads (default)
- Shared array elements are spread across the threads

```
shared int x[THREADS]       /* 1 element per thread */
shared int y[3][THREADS] /* 3 elements per thread */
shared int z[3][3]          /* 2 or 3 elements per thread */
```

- In the pictures below, assume THREADS = 4
  - Blue elts have affinity to thread 0


x
y
z
Think of linearized C array, then map in round-robin
As a 2D array, y is logically blocked by columns
z is not blocked by columns

Example, shared arrays in UPC

As a case study we can compute the constant pi using numerical integration of the following integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx = h\sum_{i=1}^{n}\frac{4}{1+x_i^2} = h\sum_{i=1}^{n}\frac{4}{1+((i-1/2)h)^2}$$

A trivial way to parallelize this is to split the sum over the threads and compute local sums that are accumulated in the end.

Create one shared array of local sums and let one thread add to a global sum.

Case Study π

```
#include <upc_relaxed.h>

shared double locsum[THREADS];
shared double pi=0;
int main(int argc, char *argv[]) {

  int i, start, stop;
  const int intervals = 100000000L ;
  double dx=1.0/intervals, x;

  start=MYTHREAD*intervals/THREADS+1;
  stop=start+intervals/THREADS-1;
  if (MYTHREAD==THREADS-1) stop=intervals;

  locsum[MYTHREAD] = 0.0;
  for (i = start; i <= stop; i++) {
    x = dx*(i - 0.5);
    locsum[MYTHREAD] += dx*4.0/(1.0 + x*x);
  }
  upc_barrier;

  if (MYTHREAD==0){
     for (i=0;i<THREADS;i++) pi+=locsum[i];
     }
  upc_barrier;
```

Blocking of shared arrays in UPC

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - **`shared int array[N]`** – (default, cyclic layout)
  - **`shared [*] int array[N]`** – (blocked, 1 block/thr)
  - **`shared [0] int array[N]`** – (all on one thread)
  - **`shared [b] int array[N]`** – (user def block size)
- Element i has affinity with thread

  `(i / block_size) % THREADS`
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping

Synchronization in UPC

UPC has several forms of barriers:

- Barrier: block until all other threads arrive

  `upc_barrier [label];`
- Split-phase barriers

  `upc_notify;` // this thread is ready for barrier

  `{compute;}` // compute unrelated to barrier

  `upc_wait;` // wait for others to be ready
- Fence construct: ensure that all shared references issued before are complete

  `upc_fence;`
- Locks for critical sections (exclusive access)

  `upc_all_lock_alloc,upc_lock_free`

  `upc_lock,upc_unlock,upc_lock_attempt`

Case Study π

```
#include <upc_relaxed.h>

shared double pi=0;
int main(int argc, char *argv[]) {

  int i, start, stop;
  const int intervals = 100000000L ;
  double dx=1.0/intervals, x,locsum;
  upc_lock_t *sum_lock = upc_all_lock_alloc();

  start=MYTHREAD*intervals/THREADS+1;
  stop=start+intervals/THREADS-1;
  if (MYTHREAD==THREADS-1) stop=intervals;

  locsum = 0.0;
  for (i = start; i <= stop; i++) {
    x = dx*(i - 0.5);
    locsum += dx*4.0/(1.0 + x*x);
  }

  upc_lock(sum_lock);
  pi+=locsum;
  upc_unlock(sum_lock);
  upc_barrier;
```

”Communication” operations in UPC

One sided read/write point-to-point:

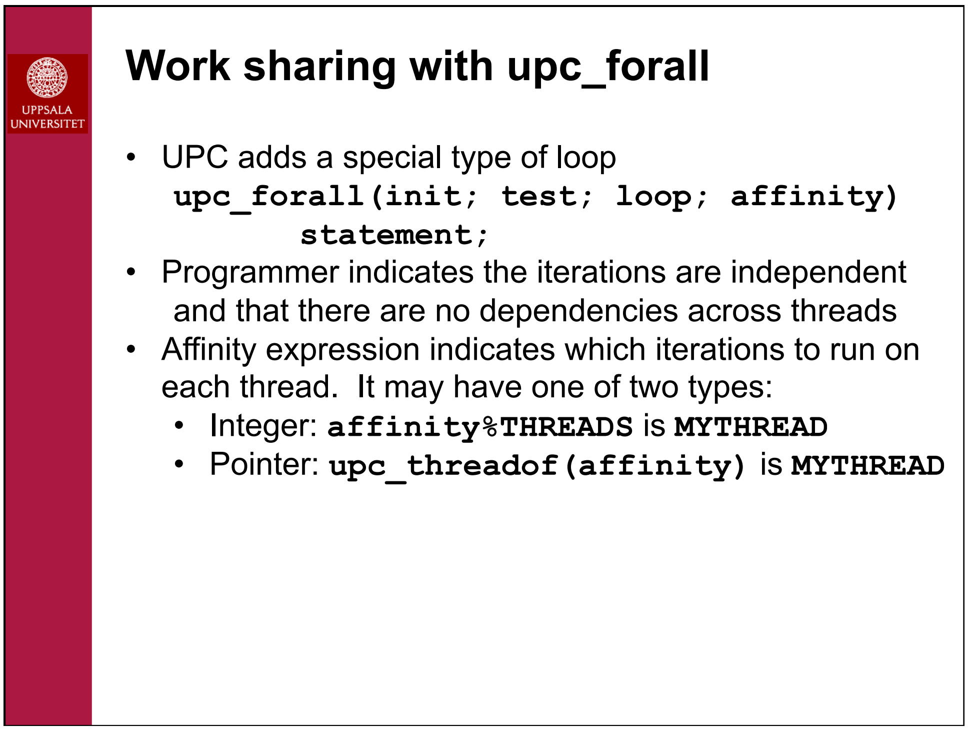
- `upc_memcpy` -- shared to shared
- `upc_memput` -- private to shared
- `upc_memget` -- shared to private

Collective operations in: `<upc_collective.h>`:

- `upc_all_broadcast`
- `upc_all_scatter`
- `upc_all_gather`
- `upc_all_gather_all`
- `upc_all_exchange`
- `upc_all_permute`
- `upc_all_reduce`
- `upc_all_prefix_reduce`

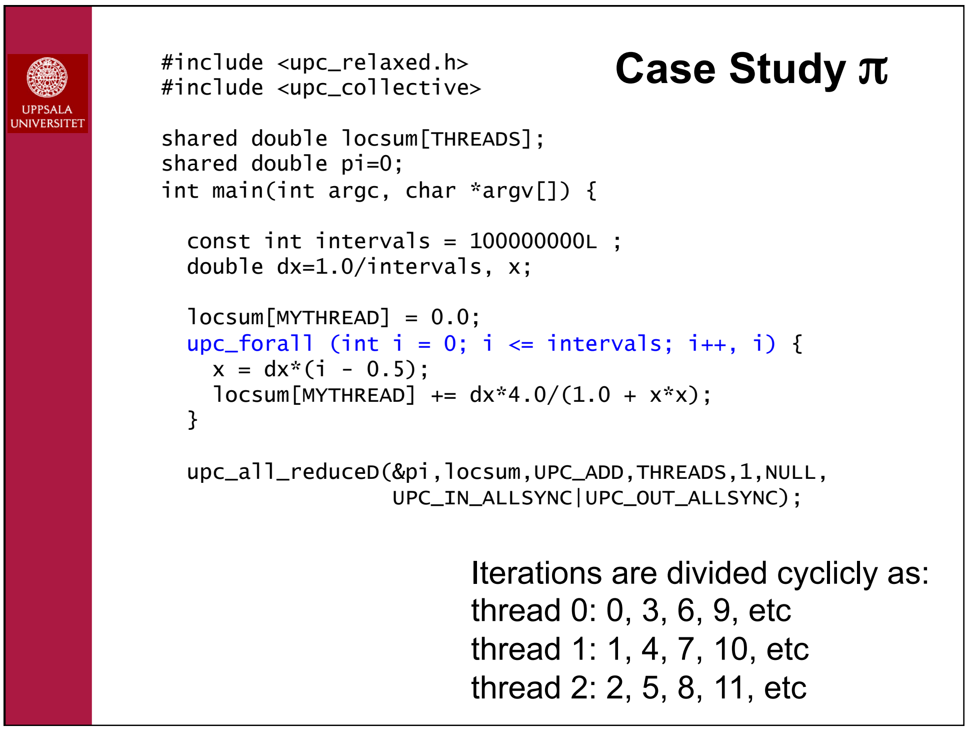
Case Study π

```
#include <upc_relaxed.h>
#include <upc_collective>

shared double locsum[THREADS];
shared double pi=0;
int main(int argc, char *argv[]) {

  int i, start, stop;
  const int intervals = 100000000L ;
  double dx=1.0/intervals, x;

  start=MYTHREAD*intervals/THREADS+1;
  stop=start+intervals/THREADS-1;
  if (MYTHREAD==THREADS-1) stop=intervals;

  locsum[MYTHREAD] = 0.0;
  for (i = start; i <= stop; i++) {
    x = dx*(i - 0.5);
    locsum[MYTHREAD] += dx*4.0/(1.0 + x*x);
  }

  upc_all_reduceD(&pi,locsum,UPC_ADD,THREADS,1,NULL,
                  UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```



Work sharing with upc_forall

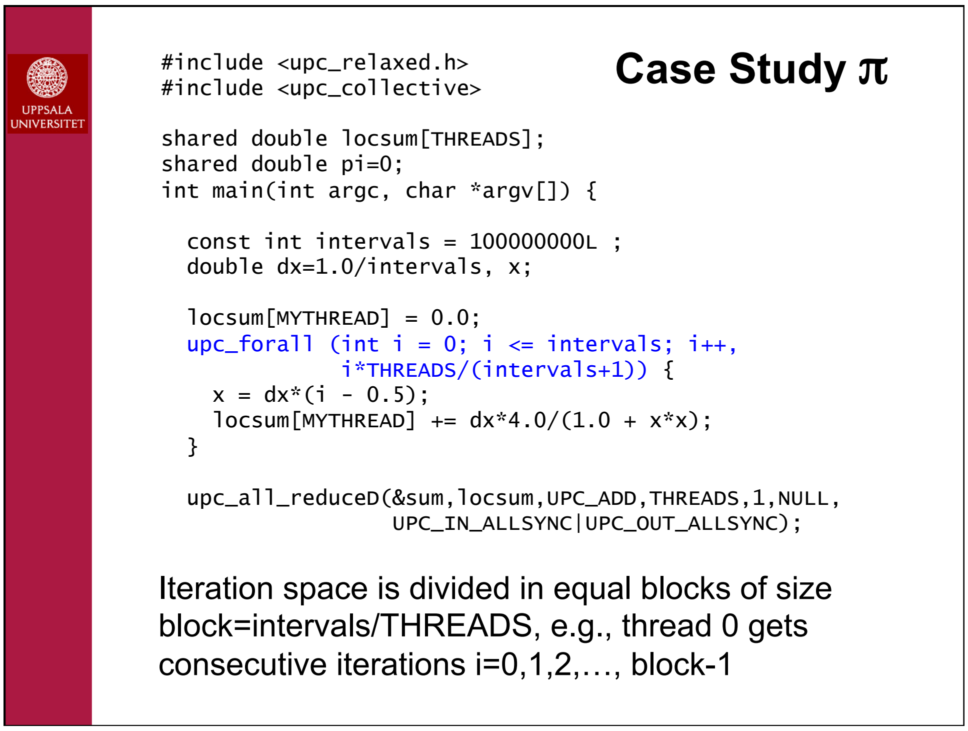
- UPC adds a special type of loop
  `upc_forall(init; test; loop; affinity)`
  `        statement;`
- Programmer indicates the iterations are independent and that there are no dependencies across threads
- Affinity expression indicates which iterations to run on each thread. It may have one of two types:
  - Integer: **`affinity%THREADS`** is **`MYTHREAD`**
  - Pointer: **`upc_threadof(affinity)`** is **`MYTHREAD`**



Case Study π

```
#include <upc_relaxed.h>
#include <upc_collective>

shared double locsum[THREADS];
shared double pi=0;
int main(int argc, char *argv[]) {

  const int intervals = 100000000L ;
  double dx=1.0/intervals, x;

  locsum[MYTHREAD] = 0.0;
  upc_forall (int i = 0; i <= intervals; i++, i) {
    x = dx*(i - 0.5);
    locsum[MYTHREAD] += dx*4.0/(1.0 + x*x);
  }

  upc_all_reduceD(&pi,locsum,UPC_ADD,THREADS,1,NULL,
                  UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

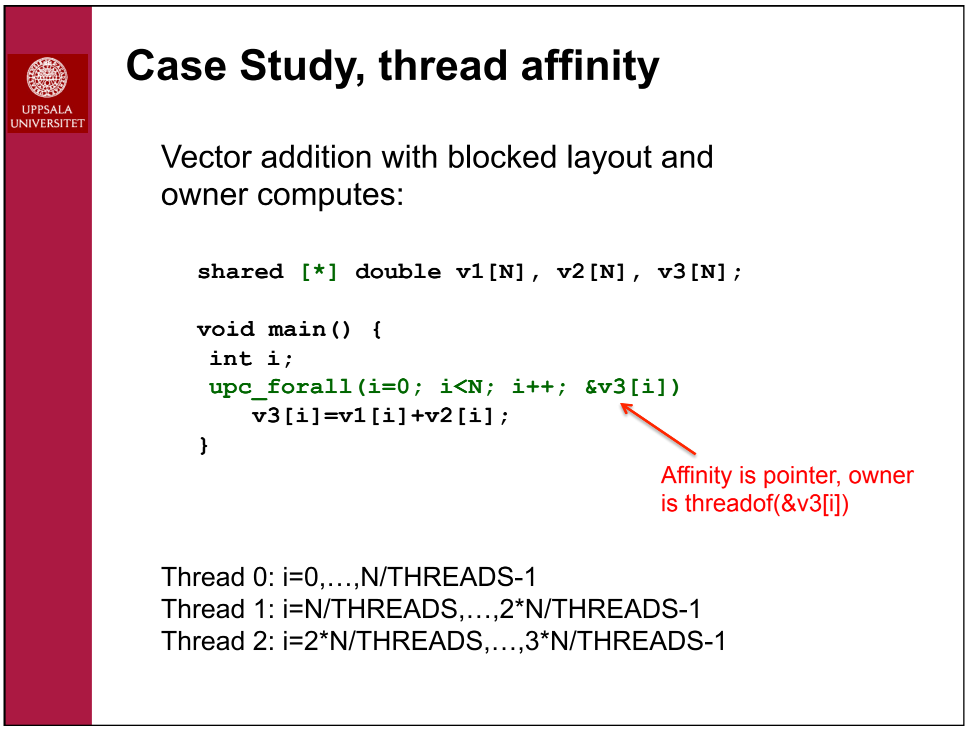
Iterations are divided cyclicly as:
thread 0: 0, 3, 6, 9, etc
thread 1: 1, 4, 7, 10, etc
thread 2: 2, 5, 8, 11, etc



Case Study π

```
#include <upc_relaxed.h>
#include <upc_collective>

shared double locsum[THREADS];
shared double pi=0;
int main(int argc, char *argv[]) {

  const int intervals = 100000000L ;
  double dx=1.0/intervals, x;

  locsum[MYTHREAD] = 0.0;
  upc_forall (int i = 0; i <= intervals; i++,
              i*THREADS/(intervals+1)) {
    x = dx*(i - 0.5);
    locsum[MYTHREAD] += dx*4.0/(1.0 + x*x);
  }

  upc_all_reduceD(&sum,locsum,UPC_ADD,THREADS,1,NULL,
                  UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

Iteration space is divided in equal blocks of size block=intervals/THREADS, e.g., thread 0 gets consecutive iterations i=0,1,2,…, block-1



Case Study, thread affinity

Vector addition with blocked layout and owner computes:

```
shared [*] double v1[N], v2[N], v3[N];

void main() {
 int i;
 upc_forall(i=0; i<N; i++; &v3[i])
     v3[i]=v1[i]+v2[i];
}
```

Affinity is pointer, owner is threadof(&v3[i])

Thread 0: i=0,…,N/THREADS-1
Thread 1: i=N/THREADS,…,2*N/THREADS-1
Thread 2: i=2*N/THREADS,…,3*N/THREADS-1

UPC Pointers

Where does the pointer point?

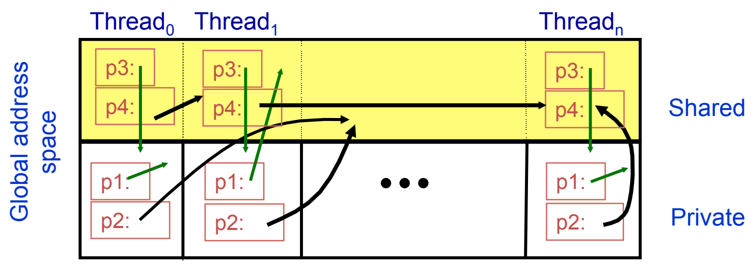
Where does the pointer reside?

	Local	Shared
Private	PP (p1)	PS (p3)
Shared	SP (p2)	SS (p4)

```
int *p1;          /* private pointer to local memory */
shared int *p2;  /* private pointer to shared space */
int *shared p3;  /* shared pointer to local memory */
shared int *shared p4; /* shared pointer to
                          shared space */
```

Shared to private is not recommended.

UPC Pointers



```
int *p1;          /* private pointer to local memory */
shared int *p2;  /* private pointer to shared space */
int *shared p3;  /* shared pointer to local memory */
shared int  *shared p4;  /* shared pointer to
                            shared space */
```

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.

UPC Pointers

```
int *p1;
```

- These pointers are fast (just like C pointers)
- Use to access local data in part of code performing local work
- Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

```
shared int *p2;
```

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication

```
int *shared p3;
```

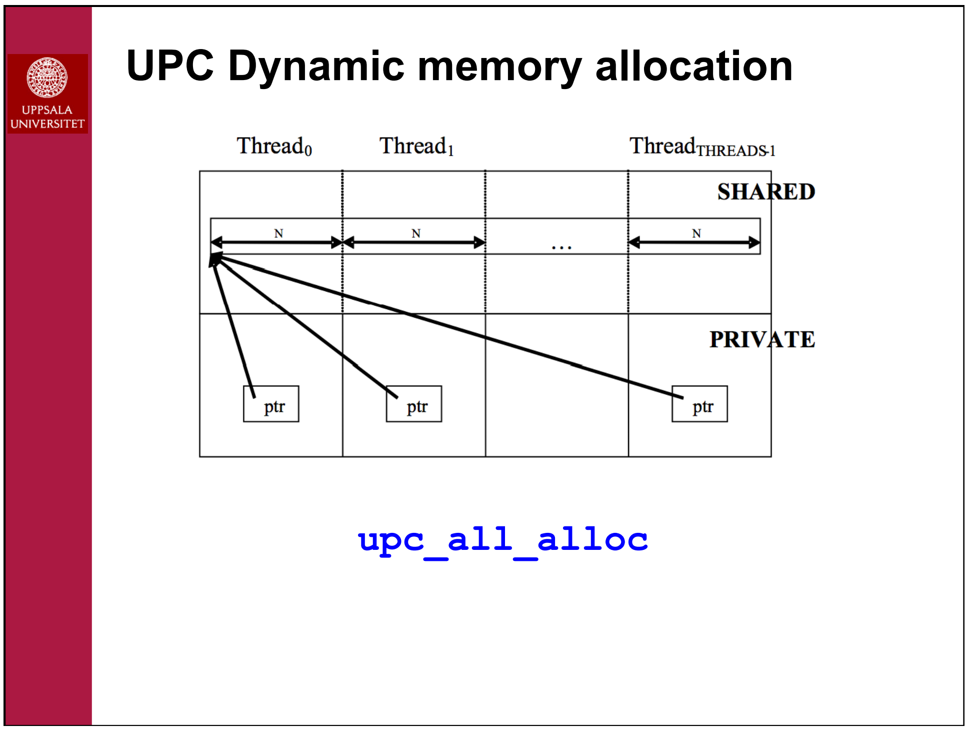
- Not recommended

```
shared int  *shared p4;
```

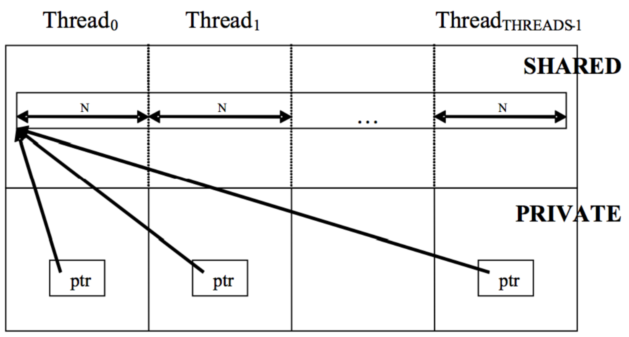
- Use to build shared linked structures, e.g., a linked list

UPC Pointers

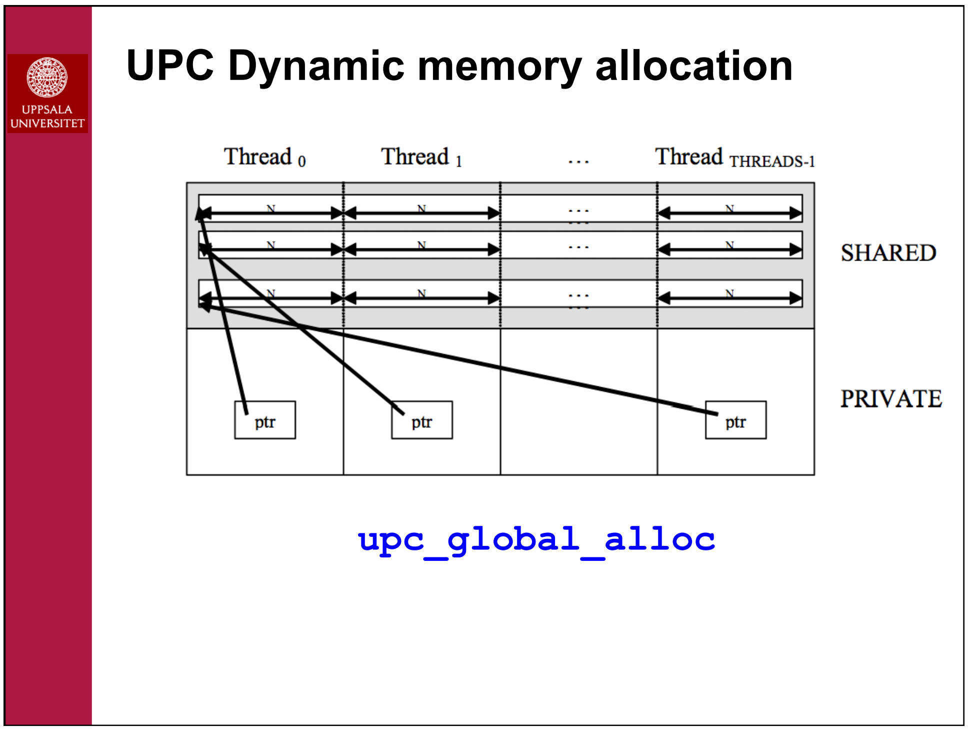
- Pointer arithmetic supports blocked and non-blocked array distributions, i.e., pointers to blocked arrays should follow the same blocking
  ```
  shared [block] double arr[N];
  shared [block] double *p=&arr[0];
  ```
- Casting of shared to private pointers is allowed but not vice versa !
- When casting a pointer-to-shared to a pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of a pointer-to-shared to a private pointer is well defined only if the pointed to object has affinity with the local thread


UPPSALA UNIVERSITET

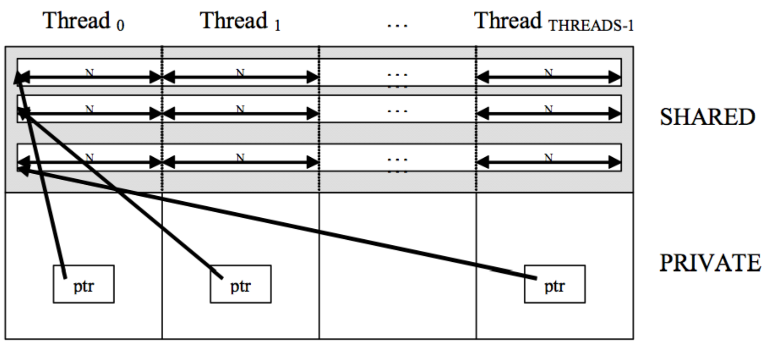
UPC Dynamic memory allocation



upc_all_alloc

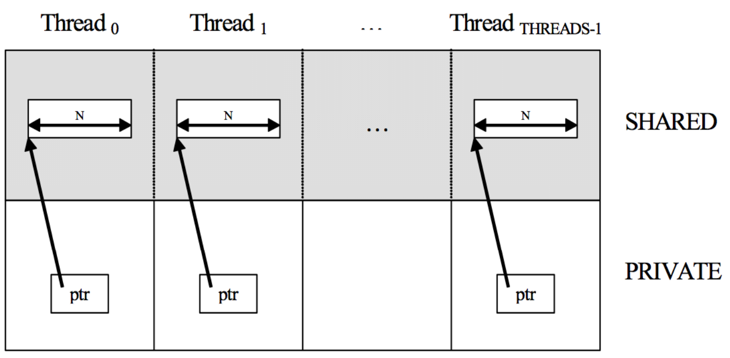

UPPSALA UNIVERSITET

UPC Dynamic memory allocation



upc_global_alloc

UPC Dynamic memory allocation

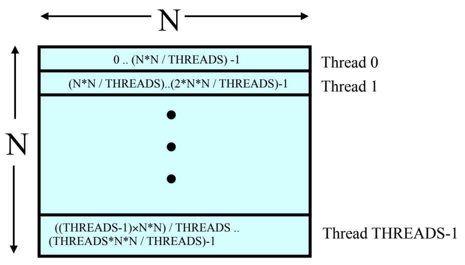


`upc_alloc`

Case Study MxM, performance

Exploit locality in matrix multiplication

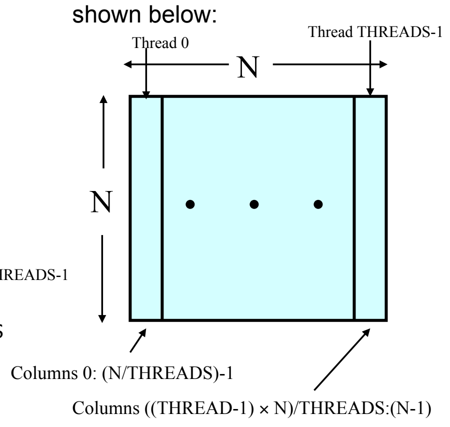
A (N × N) is decomposed row-wise into blocks of size (N×N)/THREADS as shown below:



N
N
0 .. (N*N / THREADS) -1
(N*N / THREADS)..(2*N*N / THREADS)-1
((THREADS-1)×N*N) / THREADS .. (THREADS*N*N / THREADS)-1
Thread 0
Thread 1
Thread THREADS-1

Note: N is assumed to be multiple of THREADS

B (N × N) is decomposed column wise into N/THREADS blocks as shown below:



Case Study MxM, performance

```
#define N 1024
shared [N*N/THREADS] double A[N][N];
shared [N*N/THREADS] double C[N][N];
shared [N/THREADS] double B[N][N];

int main(int argc, char *argv[]) {
    int i, j , l; // private variables

    upc_forall(i = 0 ; i<N ; i++; &C[i][0]) {
        for (j=0 ; j<N ;j++) {
            C[i][j] = 0;
            for (l= 0 ; l<N ;l++)
                C[i][j] += A[i][l]*B[l][j];
    } }
```

The UPC code for the matrix multiplication is almost the same size as the sequential code. Shared variable declarations include the keyword shared and work is parallelized with upc_forall.

Note: We are only addressing local partitions with affinity to our thread of A and C, using local pointers will speedup the code (no test-of-local).

```
#define N 1024
shared [N*N/THREADS] double A[N][N];
shared [N*N/THREADS] double C[N][N];
shared [N/THREADS] double B[N][N];

int main(int argc, char *argv[]) {
    int i, j , l; // private variables
    double *AP, *CP;

    upc_forall(i = 0 ; i<N ; i++; &C[i][0]) {
        AP=(double *)A[i]; CP=(double *)C[i];
        for (j=0 ; j<N ;j++) {
            CP[j] = 0;
            for (l= 0 ; l<N ;l++)
                CP[j] += AP[l]*B[l][j];
    } }
```

Note: B is needed in all threads, make a local copy of B

```
#define N 1024
shared [N*N/THREADS] double A[N][N];
shared [N*N/THREADS] double C[N][N];
shared [N/THREADS] double B[N][N];

int main(int argc, char *argv[]) {
    int i, j , l; // private variables
    double *AP, *CP;
    double BP[N][N];

    for( i=0; i<N; i++ )
        for( j=0; j<THREADS; j++ )
            upc_memget(&BP[i][j*(N/THREADS)],
            &B[i][j*(N/THREADS)], (N/THREADS)*sizeof(double));

    upc_forall(i = 0 ; i<N ; i++; &C[i][0]) {
        AP=(double *)A[i]; CP=(double *)C[i];
        for (j=0 ; j<N ;j++) {
            CP[j] = 0;
            for (l= 0 ; l<N ;l++)
                CP[j] += AP[l]*BP[l][j];
    } }
```

Case Study MxM

Threads	MxM, v1	MxM, v2	MxM, v3
1	42.4	24.7	8.27
2	22.4	9.97	4.46
4	8.04	3.16	2.62
8	5.55	2.30	2.67

Table: Run times MxM in UPC, N=1024, Mac, Intel Core i7, 4 physical cores

- v1 uses references to shared arrays with slow accesses
- v2 uses local references to A and C with fast accesses
- v3 makes a local copy of B which requires extra memory

UPC installations

- UPC Berkeley 2.18 installed on Tintin, Uppmax
  - module load openmpi
  - module load upc/berkeley_2.18
  - upcc –T=64 file.c // Compile for 64 proc
  - upcrun –n 64 a.out // Run
- GNU UPC installed in Tintin, Uppmax
  - module load upc/gcc_4.8
  - upc -O3 -fupc-threads-64 file.c // Compile 64 proc
  - ./a.out // Run
  - (upc –O3 file.c; ./a.out –n 64)
- Your laptop
  - Binaries for Windows and Mac OS X available at Berkeley
  - Build GNU UPC or UPC Berkeley from source files

Note, we can run UPC in parallel over several nodes with physically distributed memory on Uppmax.

UPC References

- GNU UPC
  http://www.gccupc.org
- Berkeley UPC
  http://upc.lbl.gov
- UPC Community
  http://upc.gwu.edu