Parallel Programming Languages

HPC Fall 2012

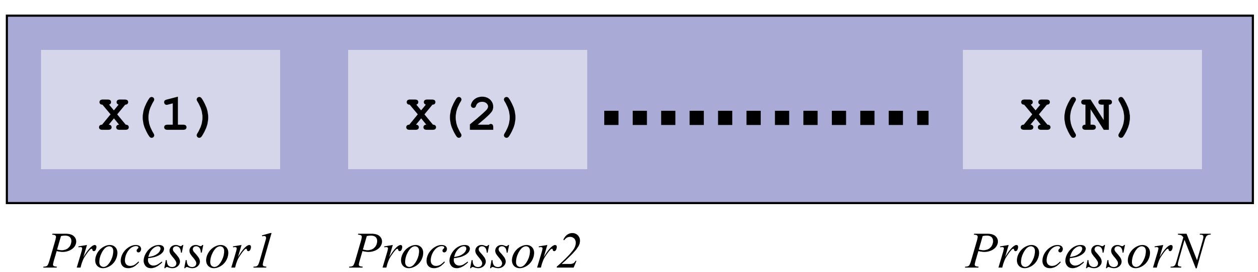
Prof. Robert van Engelen

Overview

- Partitioned Global Address Space (PGAS)
- A selection of PGAS parallel programming languages
  - CAF
  - UPC
- Further reading

Global Address Space (GAS)

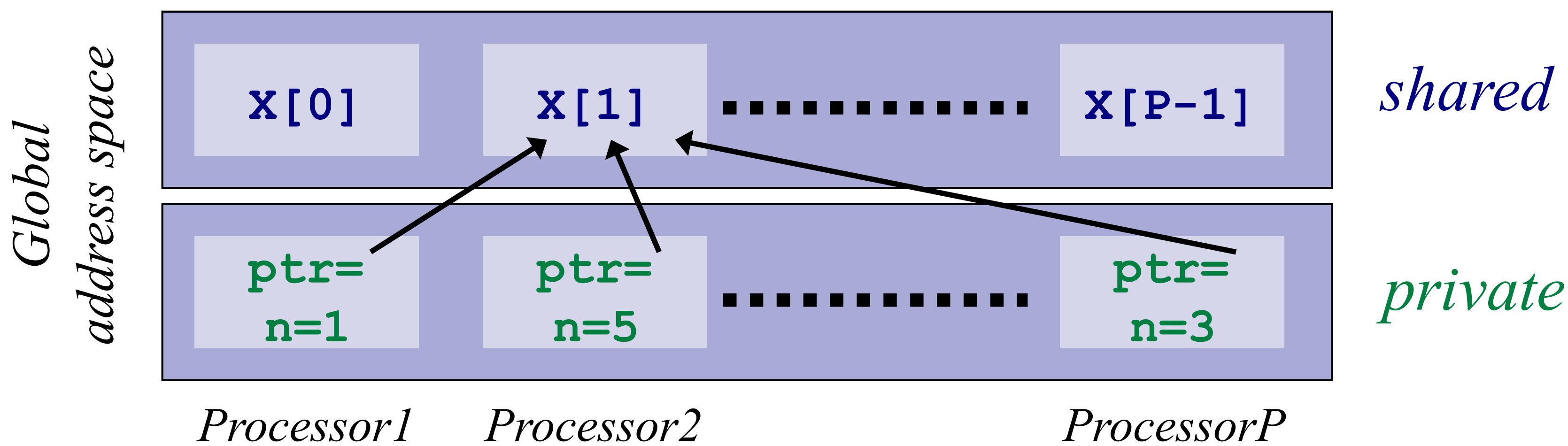
- Global address space languages take advantage of
  - Ease of programmability of shared memory parallel
  - SPMD parallelism
  - Allow local-global distinction of data, because data layout matters for performance
- Partitioned global address space is logically shared, physically distributed
  - Shared arrays are distributed over processor memories
  - Implicit communication for remote data access



Partitioned Global Address Space (PGAS)

- Global address space with two-level model that supports locality management
  - Local memory (private variables)
  - Remote memory (shared variables)

```
shared int X[P];
int *ptr = &X[1];
int n = …;
```



Partitioned Global Address Space (PGAS) Model

- Global address space with two-level memory model that supports locality management
  - Local memory (private variables)
  - Remote memory (shared variables)
- Programmer controls critical decisions
  - Data partitioning (by data placement in PGAS memory)
  - Communication (implicitly, via remote PGAS memory access)
- Suitable for mapping to a range of parallel architectures
  - Shared memory, message passing, and hybrid
- Languages: CAF (Fortran), UPC (C), X10 (Java), Titanium (Java)

PGAS Model vs Implementation

- PGAS is an abstract model
- Implementations differ with respect to details:
  - Address space partitioned by processors
    - Physically: at the memory address level (= DSM, e.g. Cray T3D/E)
    - Logically: at the variable level, where each variable can be arbitrarily placed in local memory on remote processor
  - Local caching of remote memory?
    - Coherence protocol
  - Communication
    - One-sided, e.g. DMA, is usually faster
    - Two-sided, e.g. MPI send/recv
  - Bulk memory copy operations or individual copies

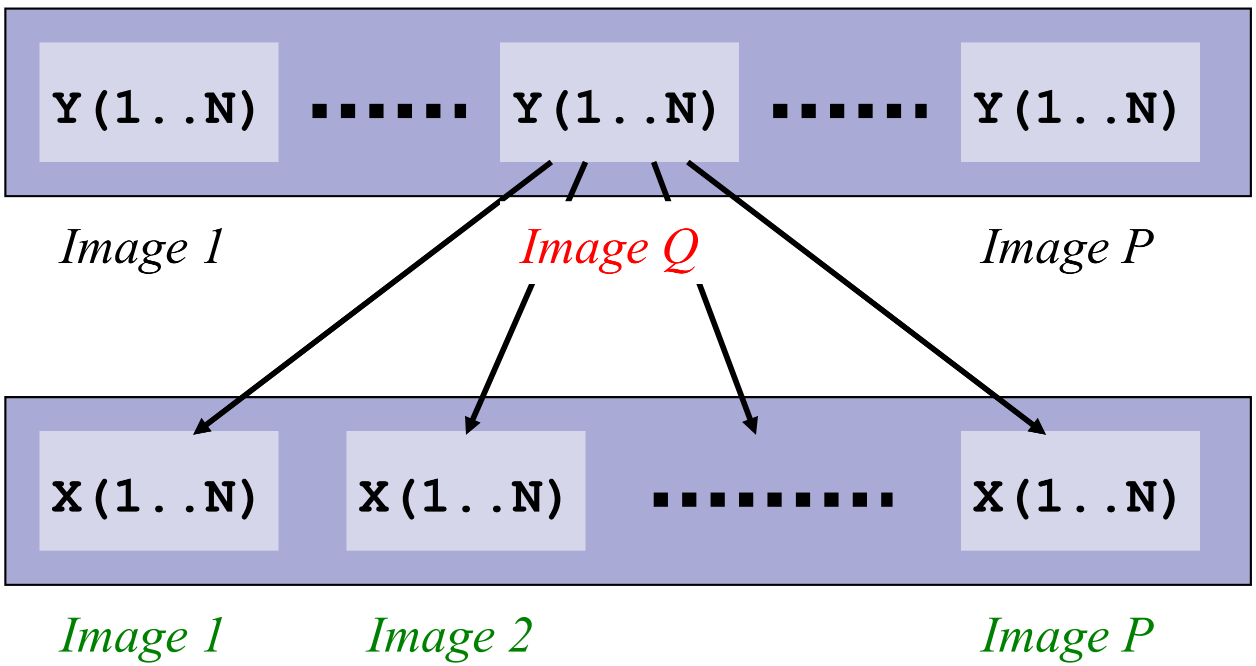
Co-Array Fortran (CAF)

- Explicitly-parallel extension of Fortran 90/95
  - Commercial compiler from Cray/SGI
  - Open source compiler from Rice University
- Partitioned global address space SPMD with two-level model that supports locality management
  - Local memory (private variables)
  - Remote memory (shared variables)
- As usual, programmer controls critical decisions
  - Data partitioning
  - Communication

CAF: Co-Arrays

- A co-array is a partitioned array with an *image* dimension

```
REAL, DIMENSION(N)[*] :: X,Y
X(:) = Y(:)[Q]
```



CAF: Array Syntax and Implicit Remote Memory Operations

```
REAL, DIMENSION(N)         :: X ! array
REAL, DIMENSION(N)[*]      :: Y ! co-array
REAL, DIMENSION(N,P)[*] :: Z ! co-array

X        = Y[PE]    ! get X(1..N) from Y(1..N)[PE]
Y[PE]    = X        ! put X(1..N) into Y(1..N)[PE]
Y[:]     = X        ! broadcast X(1..N) to Y(1..N)
Y[LIST] = X        ! broadcast X(1..N) over subset
                    !   of PEs in array LIST
Z(:)     = Y[:]     ! all-gather, collect Y(1..N)
                    !   over PEs in Z(1..N,1..P)
S = MINVAL(Y[:]) ! min (reduce) Y(1..N) over PEs
Z[:]     = S        ! S scalar, promoted to array
                    !   of shape (1:N,1:P)
```

CAF: Synchronization

```
COMMON/XCTILB4/ B(N,4)[*]
SAVE   /XCTILB4/

ME = THIS_IMAGE()
IF (ME > 1 .AND. ME < NUM_IMAGES()) THEN
  CALL SYNC_ALL( WAIT=(/ME-1,ME+1/) )
  B(:,1) = B(:,3)[ME-1]
  B(:,4) = B(:,2)[ME+1]
  CALL SYNC_ALL( WAIT=(/ME-1,ME+1/) )
ENDIF
```

Wait for processors on the left and right

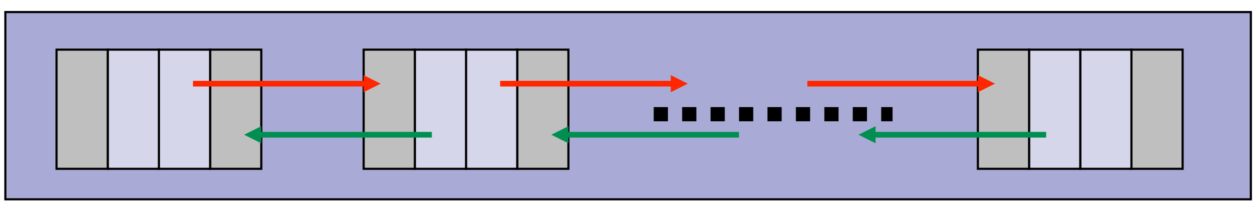


Image 1 *Image 2* *Image P*

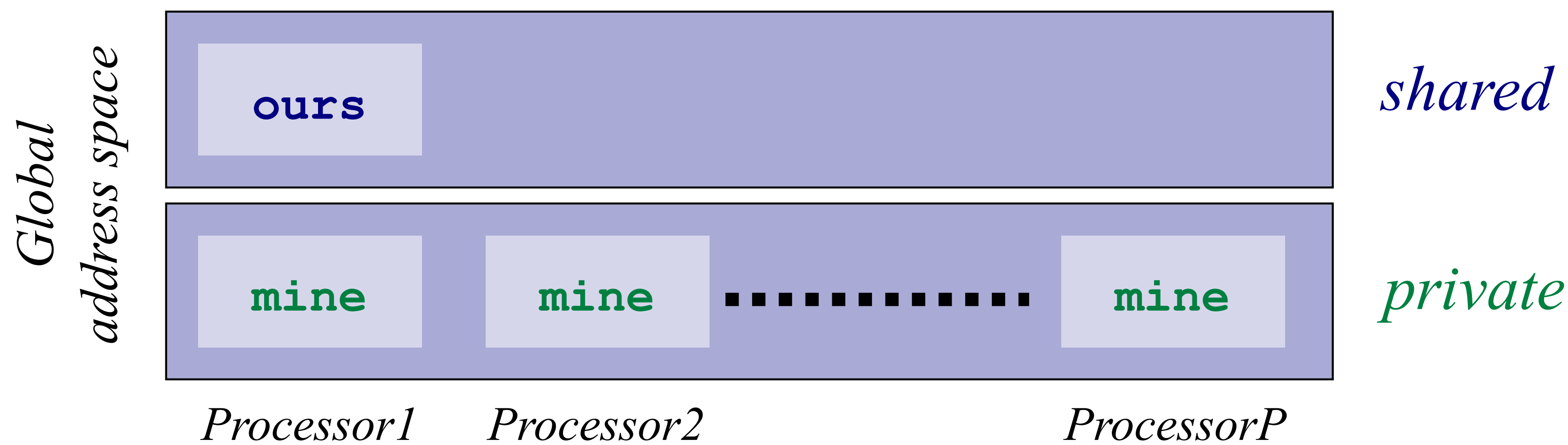
Unified Parallel C (UPC)

- UPC is an explicit extension of ANSI C
  - Commercial compilers from Cray/SGI, HP
  - Open source compiler from LBNL/UCB/MTU/UF and GCC-UPC project
- Follows the C language philosophy
  - Programmers are clever and careful and may need to work close to the hardware level
    - to get performance,
    - but allows you to get into trouble, just like programming low level C!
  - Concise and efficient syntax
- UPC is a PGAS language
  - Global address space with private and shared variables
  - Private/shared pointers to private/shared variables
  - Array data distributions (block/cyclic)
  - Forall worksharing loops
  - Barriers and locks
  - Bulk copy operations between shared and private memory

UPC: Shared Variables

- Private by default
  - C variables and objects are allocated in private memory space for each thread
- Shared variables are explicitly declared and allocated once (by thread 0)
  - Shared variables must be "globally" declared (i.e. static)

```
shared int ours;
int mine;
```

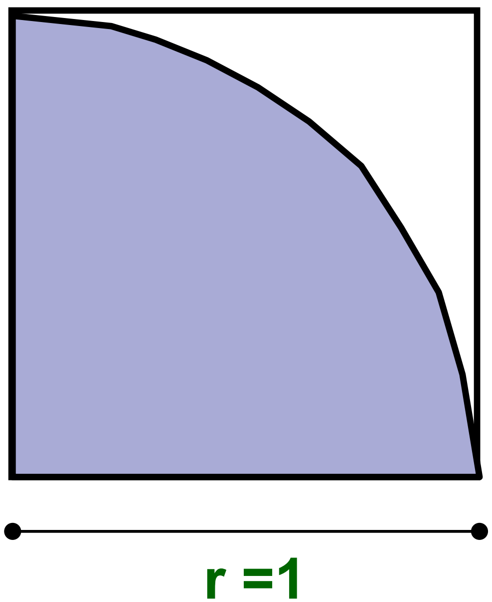


UPC: Simple Example Monte Carlo pi Calculation

```
int hit()
{
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  return ((x*x + y*y) <= 1.0);
}
```

Randomly throw darts at (x,y) positions in a unit circle, if $x^2 + y^2 \leq 1$, then point is inside circle

Compute ratio of points inside/total, then $\pi = 4*ratio$



UPC: Simple Example Monte Carlo pi Calculation

```
#include <upc.h>
shared int hits = 0;
main()
{ int i;
  int my_trials, trials = …;
  my_trials = (trials + THREADS - 1)/THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    hits += hit();
  if (MYTHREAD == 0)
    printf("pi estimated to %g\n",
           4*(double)hits/(double)trials);
}
```

Divide the work

Score hits

What can go wrong?

UPC: Simple Example Monte Carlo pi Calculation

```
shared int hits = 0;
main()
{ int i, my_trials, trials = …;
  upc_lock_t *hit_lock = upc_all_lock_alloc();
  my_trials = (trials + THREADS - 1)/THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
  { upc_lock(hit_lock);
    hits += hit();
    upc_unlock(hit_lock);
  }
  upc_barrier;
  if (MYTHREAD == 0)
    printf("pi estimated to %g\n",
           4*(double)hits/(double)trials);
  upc_lock_free(hit_lock);
}
```

Score hits

Synchronize

Anything wrong here…?

UPC: Simple Example Monte Carlo pi Calculation

```
shared int hits[THREADS] = { 0 };
main()
{ int i, my_trials, trials = …;
  my_trials = (trials + THREADS - 1)/THREADS;
  srand(MYTHREAD*17);
  for (i=0; i < my_trials; i++)
    hits[MYTHREAD] += hit();
  upc_barrier;
  if (MYTHREAD == 0)
  { for (i=1; i < THREADS; i++)
      hits[0] += hits[i];
    tot_trials = THREADS*my_trials;
    printf("pi estimated to %g\n",
           4*(double)hits[0]/(double)tot_trials);
  }
}
```

Score hits

Sync

Sum hits

Corrected

UPC: Forall Work Sharing

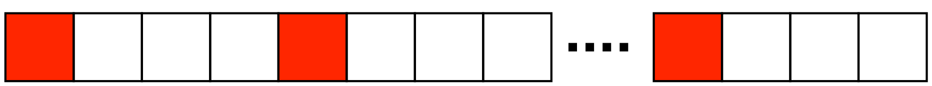
```
shared int v1[N], v2[N], sum[N];

int i;
upc_forall(i=0; i<N; i++; i)
  sum[i] = v1[i] + v2[i];
```

Default distribution: cyclic

Affinity: here it forces *<u>owner-computes</u> rule*

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

Assume **THREADS**=4



Elements with affinity to processor 0 are shown in red

UPC: Pointers

Where does the pointer reside?

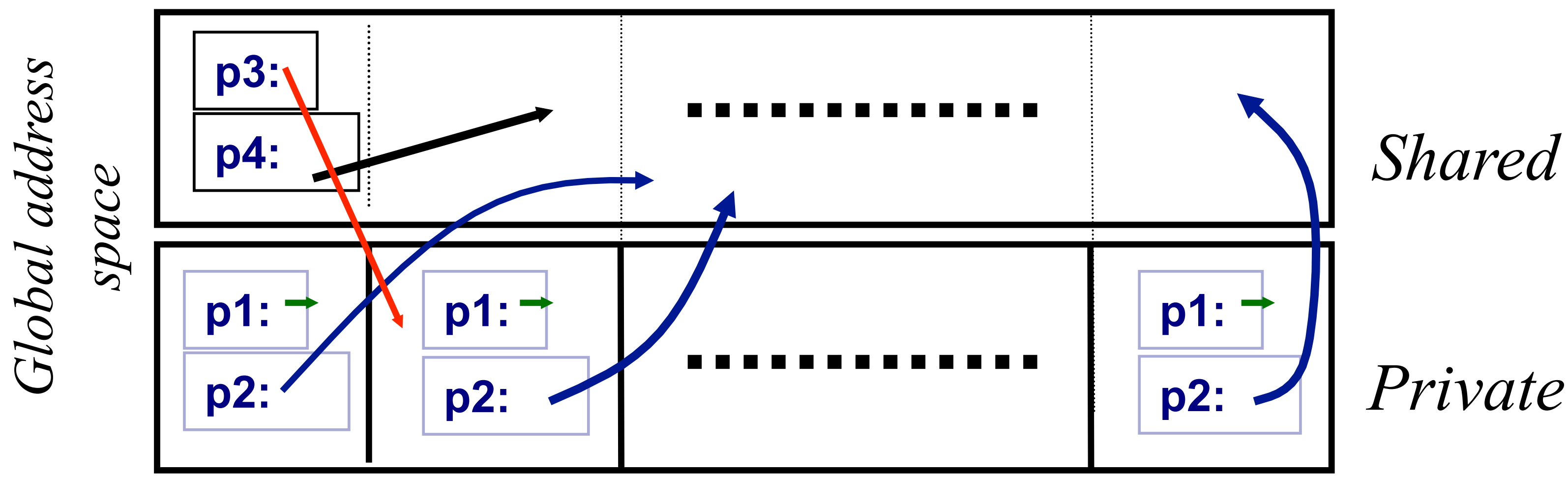
Where does the referenced value reside?

	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 pointer to private local memory 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 */
```

UPC: Pointer Example

```
shared int v1[N], v2[N], sum[N];

int i;
shared int *p1, *p2;

p1 = v1;
p2 = v2;
upc_forall(i=0; i<N; i++, p1++, p2++; i)
  sum[i] = *p1 + *p2;
```

UPC: Pointers

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block (for blocked data distributions)
  - phase (specifies position in the block)

Phase	Thread	Virtual Address
63 49	48 38	37 0

UPC: Shared Variable Layout

- Non-array shared variables have affinity with thread 0
- Array layouts are cyclic or blocked:

```
shared double x[n];       /* cyclic */
shared [b] double y[n];   /* blocked */
```

  where **b** is the block size

- For blocked layouts, element *i* has affinity with thread:

```
(i/b) % THREADS
```

  therefore use **i/b** in forall (owner-computes):

```
upc_forall(i=0; i<N; i++; i/b) y[i] = …
```

UPC: Consistency Model

- The consistency model of shared memory accesses are controlled by qualifiers
  - Strict: will always appear in order
  - Relaxed: may appear out of order to other threads

```
strict: {
  x = y;
  z = y+1;
}
```

- Use strict on variables that are used as synchronization

Thread1	*Thread2*
`flag = 0;`	`while (flag)`
`data = …;`	`  … = data;`
`flag = 1;`	

- Select the default consistency model with:
  - `#include <upc_strict.h>`
  - `#include <upc_relaxed.h>`

UPC: Fence

- UPC provides a fence construct
  - Syntax

    ```
    upc_fence;
    ```

    equivalent to a null strict reference

    ```
    strict { }
    ```

  - Ensures that all shared references issued before the **`upc_fence`** are complete

Further Reading

- CAF: www.co-array.org
- UPC: upc.gwu.edu