

Lecture 18: Unified Parallel C

David Bindel

31 Mar 2010

References

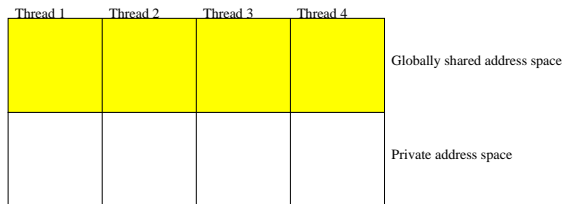
- ▶ <http://upc.lbl.gov>
- ▶ <http://upc.gwu.edu>

Based on slides by Kathy Yelick (UC Berkeley),
in turn based on slides by Tarek El-Ghazawi (GWU)

Big picture

- ▶ Message passing: scalable, harder to program (?)
- ▶ Shared memory: easier to program, less scalable (?)
- ▶ Global address space:
 - ▶ Use shared address space (programmability)
 - ▶ Distinguish local/global (performance)
 - ▶ Runs on distributed or shared memory hw

Partitioned Global Address Space (PGAS)



- ▶ Partition a shared address space:
 - ▶ *Local* addresses live on local processor
 - ▶ *Remote* addresses live on other processors
 - ▶ May also have *private* address spaces
 - ▶ Programmer controls data placement
- ▶ Several examples: UPC, Co-Array Fortran, Titanium

Unified Parallel C

Unified Parallel C (UPC) is:

- ▶ Explicit parallel extension to ANSI C
- ▶ A partitioned global address space language
- ▶ Similar to C in design philosophy: concise, low-level, ... and “enough rope to hang yourself”
- ▶ Based on ideas from Split-C, AC, PCP

Execution model

- ▶ THREADS parallel threads, MYTHREAD is local index
- ▶ Number of threads can be specified at compile or run-time
- ▶ Synchronization primitives (barriers, locks)
- ▶ Parallel iteration primitives (forall)
- ▶ Parallel memory access / memory management
- ▶ Parallel library routines

Hello world

```
#include <upc.h> /* Required for UPC extensions */
#include <stdio.h>

int main()
{
    printf("Hello from %d of %d\n",
          MYTHREAD, THREADS);
}
```

Shared variables

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

- ▶ Normal variables allocated in private memory per thread
- ▶ Shared variables allocated once, on thread 0
- ▶ Shared variables cannot have dynamic lifetime
- ▶ Shared variable access is more expensive

Shared arrays

```
shared int x[THREADS];      /* 1 per thread */  
shared double y[3*THREADS]; /* 3 per thread */  
shared int z[10];          /* Varies */
```

- ▶ Shared array elements have *affinity* (where they live)
- ▶ Default layout is cyclic
 - ▶ e.g. $y[i]$ has affinity to thread $i \% \text{THREADS}$

Hello world++ = π via Monte Carlo

Write

$$\pi = 4 \frac{\text{Area of unit circle quadrant}}{\text{Area of unit square}}$$

If (X, Y) are chosen uniformly at random on $[0, 1]^2$, then

$$\pi/4 = P\{X^2 + Y^2 < 1\}$$

Monte Carlo calculation of π : sample points from the square and compute fraction that fall inside circle.

π in C

```
int main()
{
    int i, hits = 0, trials = 1000000;
    srand(17); /* Seed random number generator */
    for (i = 0; i < trials; ++i)
        hits += trial_in_disk();
    printf("Pi approx %g\n", 4.0*hits/trials);
}
```

π in UPC, Version 1

```
shared int all_hits[THREADS];
int main() {
    int i, hits = 0, tot = 0, trials = 1000000;
    srand(1+MYTHREAD*17);
    for (i = 0; i < trials; ++i)
        hits += trial_in_disk();
    all_hits[MYTHREAD] = hits;
    upc_barrier;
    if (MYTHREAD == 0) {
        for (i = 0; i < THREADS; ++i)
            tot += all_hits[i];
        printf("Pi approx %g\n", 4.0*tot/trials/THREADS);
    }
}
```

Synchronization

- ▶ Barriers: `upc_barrier`
- ▶ Split-phase barriers: `upc_notify` and `upc_wait`
`upc_notify;`
Do some independent work
`upc_wait;`
- ▶ Locks (to protect critical sections)

Locks

Locks are dynamically allocated objects of type `upc_lock_t`:

```
upc_lock_t* lock = upc_all_lock_alloc();  
upc_lock(lock);      /* Get lock */  
upc_unlock(lock);    /* Release lock */  
upc_lock_free(lock); /* Free */
```

π in UPC, Version 2

```
shared int tot;
int main() {
    int i, hits = 0, trials = 1000000;
    upc_lock_t* tot_lock = upc_all_lock_alloc();
    srand(1+MYTHREAD*17);
    for (i = 0; i < trials; ++i)
        hits += trial_in_disk();
    upc_lock(tot_lock);
    tot += hits;
    upc_unlock(tot_lock);
    upc_barrier;
    if (MYTHREAD == 0) { upc_lock_free(tot_lock); print ...}
}
```

Collectives

UPC also has collective operations (typical list)

```
#include <bupc_collectivev.h>
int main() {
    int i, hits = 0, trials = 1000000;
    srand(1+MYTHREAD*17);
    for (i = 0; i < trials; ++i)
        hits += trial_in_disk();
    hits = bupc_allv_reduce(int, hits, 0, UPC_ADD);
    if (MYTHREAD == 0) printf(...);
}
```


Loop parallelism with `upc_forall`

UPC adds a special type of extended for loop:

```
upc_forall(init; test; update; affinity)
    statement;
```

- ▶ Assume no dependencies across threads
- ▶ Just run iterations that match affinity expression
 - ▶ Integer: `affinity % THREADS == MYTHREAD`
 - ▶ Pointer: `upc_threadof(affinity) == MYTHREAD`
- ▶ Really syntactic sugar (could do this with `for`)

Example

Note that x , y , and z all have the same layout.

```
shared double x[N], y[N], z[N];
int main() {
    int i;
    upc_forall(i=0; i < N; ++i; i)
        z[i] = x[i] + y[i];
}
```

Array layouts

- ▶ Sometimes we don't want cyclic layout (think nearest neighbor stencil...)
- ▶ UPC provides *layout specifiers* to allow block cyclic layout
- ▶ Block sizes expressions must be compile time constant (except THREADS)
- ▶ Element i has affinity with $(i / \text{blocksize}) \% \text{THREADS}$
- ▶ In higher dimensions, affinity determined by linearized index

Array layouts

Examples:

```
shared double a[N];      /* Block cyclic */
shared[*] double a[N];  /* Blocks of N/THREADS */
shared[] double a[N];   /* All elements on thread 0 */
shared[M] double a[N];  /* Block cyclic, block size M */
shared[M1][M2] double a[N][M1][M2]; /* Blocks of M1*M2 */
```

1D Jacobi Poisson example

```
shared[*] double u_old[N], u[N], f[N]; /* Block layout */
void jacobi_sweeps(int nsweeps) {
    int i, it;
    upc_barrier;
    for (it = 0; it < nsweeps; ++it) {
        upc_forall(i=1; i < N-1; ++i; &(u[i]))
            u[i] = (u_old[i-1] + u_old[i+1] - h*h*f[i])/2;
        upc_barrier;
        upc_forall(i=0; i < N; ++i; &(u[i]))
            u_old[i] = u[i];
        upc_barrier;
    }
}
```

1D Jacobi pros and cons

Good points about Jacobi example:

- ▶ Simple code (1 slide!)
- ▶ Block layout minimizes communication

Bad points:

- ▶ Shared array access is relatively slow
- ▶ Two barriers per pass

1D Jacobi: take 2

```
shared double ubound[2][THREADS]; /* For ghost cells*/
double uold[N_PER+2], uloc[N_PER+2], floc[N_PER+2];
void jacobi_sweep(double h2) {
    int i;
    if (MYTHREAD>0)          ubound[1][MYTHREAD-1]=uold[1];
    if (MYTHREAD<THREADS) ubound[0][MYTHREAD+1]=uold[N_PER];
    upc_barrier;
    uold[0]          = ubound[0][MYTHREAD];
    uold[N_PER+1] = ubound[1][MYTHREAD];
    for (i = 1; i < N_PER+1; ++i)
        uloc[i] = (uold[i-1] + uold[i+1] - h2*floc[i])/2;
    for (i = 1; i < N_PER+1; ++i)
        uold[i] = uloc[i];
}
```

1D Jacobi: take 3

```
void jacobi_sweep(double h2) {
    int i;
    if (MYTHREAD>0)          ubound[1][MYTHREAD-1]=uold[1];
    if (MYTHREAD<THREADS) ubound[0][MYTHREAD+1]=uold[N_PER];
    upc_notify; /***** Start split barrier *****/
    for (i = 2; i < N_PER; ++i)
        uloc[i] = (uold[i-1] + uold[i+1] - h2*floc[i])/2;
    upc_wait; /***** End split barrier *****/
    uold[0]          = ubound[0][MYTHREAD];
    uold[N_PER+1]   = ubound[1][MYTHREAD];
    for (i = 1; i < N_PER+1; i += N_PER)
        uloc[i] = (uold[i-1] + uold[i+1] - h2*floc[i])/2;
    for (i = 1; i < N_PER+1; ++i) uold[i] = uloc[i];
}
```


Sharing pointers

Have pointers to global address space. Either pointer or referenced data might be shared:

```
int* p;                /* Ordinary pointer */
shared int* p;         /* Local pointer to shared data */
shared int* shared p; /* Shared pointer to shared data */
int* shared p;         /* Legal, but bad idea */
```

Pointers to shared are larger and slower than standard pointers.

UPC pointers

Pointers to shared objects have three fields:

- ▶ Thread number
- ▶ Local address of block
- ▶ Phase (position in block)

Access with `upc_threadof` and `upc_phaseof`;
go to start with `upc_resetphase`.

Dynamic allocation

- ▶ Can dynamically allocate shared memory
- ▶ Functions can be collective or not
- ▶ Collective functions must be called by every thread, return same value at all threads

Global allocation

```
shared void*  
upc_global_alloc(size_t nblocks, size_t nbytes);
```

- ▶ Non-collective – just called at one thread
- ▶ Layout of shared `[nbytes] char[nblocks * nbytes]`

Collective global allocation

```
shared void*  
upc_all_alloc(size_t nblocks, size_t nbytes);
```

- ▶ Collective – everyone calls, everyone receives same pointer
- ▶ Layout of shared [nbytes] char[nblocks * nbytes]

UPC free

```
void upc_free(shared void* p);
```

- ▶ Frees dynamically allocated shared memory
- ▶ *Not* collective

Example: Shared integer stack

Shared linked-list representation of a stack (think work queues).
All data will be kept at thread 0.

```
typedef struct list_t {
    int x;
    shared struct list_t* next;
} list_t;

shared struct list_t* shared head;
upc_lock_t* list_lock;
```

Example: Shared integer stack

```
void push(int x) {
    shared list_t* item =
        upc_global_alloc(1, sizeof(list_t));
    upc_lock(list_lock);
    item->x = x;
    item->next = head;
    head = item;
    upc_unlock(list_lock);
}
```


Example: Shared integer stack

```
int pop(int* x) {
    shared list_t* item;
    upc_lock(list_lock);
    if (head == NULL) {
        upc_unlock(list_lock);
        return -1;
    }
    item = head;
    head = head->next;
    *x = item->x;
    upc_free(item);
    upc_unlock(list_lock);
    return 0;
}
```

Memory consistency

UPC has two types of accesses:

- ▶ Strict: will always appear in order (sequential consistency)
- ▶ Relaxed: may appear out of order to other threads

Several ways to specify:

- ▶ Include `<upc_relaxed.h>`
- ▶ Add `strict` or `relaxed` as type qualifier
- ▶ Use pragmas

The `upc_fence` is a strict null reference – ensures shared references issued earlier are complete.

Performance

People won't use it if it's too slow! So:

- ▶ Maximize single-node performance (can link with tuned libraries, build on fast compilers)
- ▶ Use fast communication (GASNet layer provides fast one-sided communication for Berkeley UPC)
- ▶ Manage the details intelligently (language provides access to some low-level details, such as memory layout).

Case studies as part of UPC tutorial slides. With care, can sometimes get better performance than MPI!

But performance tuning is still nontrivial... not a magic bullet.