Unified Parallel C

Abed Haque
Brice Ngnigha

Overview

- Unified parallel C UPC is an extension of the C programming language designed for high-performance computing on large-scale parallel machines.
- Started at University of California-Berkeley in 1996 and now is a community effort. Based on Split-C UCB, AC (IDA), PCP (LLNL)
- We will discuss
  - The Installation
  - The Execution Model
  - The Memory Model
  - The Threads Synchronization
  - The Dynamic Memory Management
  - The language Keywords

References

- UPC language specifications
  https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.2.pdf
- GNU UPC installation sources
  - Berkeley UPC: http://www.gccupc.org/
  - GNU UPC: http://upc.lbl.gov/download/
- http://upc.gwu.edu/
  - The George Washington University - The High Computing Performance Laboratory
- Webinar: Partitioned Global Address Space Programming with UPC
  https://youtu.be/Ey-inJ9Dz6Q?t=904

Installation

Multiple implementations exist out there, each supporting specific platforms and providing their unique way to install.

- Berkeley UPC: Provides source package for Windows and MacOS along with installation instructions in order to build the compiler.
- GUPC (GNU UPC): Provides dependencies, installation and compilation instructions for Linux builds, Cray XT3/4/5, Mac OS Yosemite build.
- Releases for specific platforms are also available from these sources

UPC Execution Model

- Follows the Single Program Multiple Data model (SPMD)
- Requires a fixed number of threads THREADS
- Each thread receive a thread ID at creation accessible in the program as MYTHREAD
- Two modes for passing THREADS:
  - Static thread mode: THREADS specified at compilation
  - Dynamic thread mode: Compiled code executed with an option for number of THREADS
- Each thread gets its own copy of the local variables
- Each thread can read the input arguments

UPC Execution Model

- Basic Hello World UPC

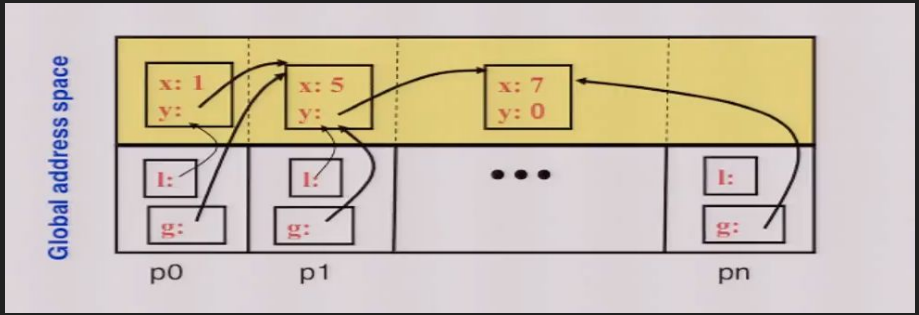
```
#include <stdio.h>
#include <upc.h>

int main( int argc, char **argv) {
    printf(Hello world: this is thread %d of %d\n”, MYTHREAD, THREADS);
    return 0;
}
```

- Output

```
abed@ubuntu:~/Desktop/upc_samples$ upc hello.upc -o hello
abed@ubuntu:~/Desktop/upc_samples$ ./hello -fupc-threads-5
Hello world: this is thread 1 of 5
Hello world: this is thread 4 of 5
Hello world: this is thread 0 of 5
Hello world: this is thread 2 of 5
Hello world: this is thread 3 of 5
```

UPC Memory Model

- Partitioned Global Addressing Space PGAS model
  - Partitioned: Data is designated as local or global.
  - Global Space Addressing: Threads may directly read or write remote.



UPC Memory Model

- Uses both the Shared memory and the Private memory
- Shared Memory model
  - Uses keyword “shared” to declare a variable in the shared space
  - Shared variables are allocated once by thread 0
  - Shared objects cannot be declared inside of functions
  - Shared arrays are distributed over the threads in a circular fashion
  - Shared arrays are great to prevent race conditions
- Private memory model
  - Just like the C language, variables are allocated in the private memory space of each thread

Threads Synchronization

The language uses many synchronization techniques similar to MPI

- Barriers: “upc_barrier” globally blocks until every threads arrive.
- Split-Phase Barriers:
  - “upc_notify”: this thread notifies that it is ready to sync. Can move on to do unrelated computation.
  - “upc_wait”: after notifying, this thread can do some work and when done, calls upc_wait to wait until remaining threads call upc_notify.
  - Usage:

```
upc_notify;
if MYTHREAD == someID
    /* Do some unrelated work */
Upc_wait;
```

Threads Synchronization

- Labels: To synchronize threads that have reached a condition and may execute on different execution paths.

```
#define A_MERGE_LABEL 5
if(a)
     upc_Barrier A_MERGE_LABEL;
else if(b)
     upc_Barrier A_MERGE_LABEL;
```

- Locks: represented by an opaque structure “upc_lock_t”
  - Locks are to be allocated before use
  - upc_lock_t *upc_all_lock_alloc(void) allocates 1 lock, all threads point to it
  - Upc_lock_t *upc_global_lock_alloc(void) allocates 1 lock, each thread own the allocated lock
  - void upc_lock(upc_lock_t *l)
  - void upc_unlock(upc_lock_t *l)
  - void upc_lock_free (upc_lock_t *ptr)

Dynamic Memory management

- Non-collective Shared space memory allocation
  - shared void *upc_alloc()
- Collective Shared space memory allocation
  - shared void *upc_all_alloc()
- Memory allocation functions can also be used to allocate private memory space can also be used just like in C language

UPC Pointers and Data

UPC provides multiple memory addressing schemes

- Private pointer can point to local memory
  - `int *p1`
- Private pointer can point to shared space
  - `shared in *p2`
- Shared pointer to local memory
  - `int *shared p3` (not recommended)
- Shared pointer to shared space
  - `shared int *shared p4`
- UPC pointers to shared objects have 3 fields
  - Thread number, local address of block, phase (position in block)

UPC Pointers and Data

- The `shared` keyword

```
int x;                  // each thread has own copy
shared int y;           // shared copy in Thread 0
shared int z[THREADS];  // 1 element per thread
```

Thread 0	Thread 1	Thread 2
x	x	x
y		
z[0]	z[1]	z[2]

UPC Pointers and Data

- `shared` 2d arrays

```
shared int matrix[4][THREADS];    // Assume 3 threads
```

The 2d array `matrix` is logically blocked by columns

Thread 0	Thread 1	Thread 2
`matrix[0][0]`	`matrix[0][1]`	`matrix[0][2]`
`matrix[1][0]`	`matrix[1][1]`	`matrix[1][2]`
`matrix[2][0]`	`matrix[2][1]`	`matrix[2][2]`
`matrix[3][0]`	`matrix[3][1]`	`matrix[3][2]`

UPC Pointers and Data

- `shared` 2d arrays
  Block sizes can be specified (the default block size is 1)
  Syntax: `shared [block-size] type array [N];`
  Ex: `shared [3] int A[4][THREADS]; // Assume THREADS = 4`

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

UPC Looping Mechanism

- `upc_forall:` Mechanism to alleviate the cost of the For loop in SPMD programs.
  - Basic for loop in C, Java...

```
for( i = 0; i < N; i++ )
    if( i == MYTHREAD ) do { /* Do something */}
```

  - Using upc_forall

```
ucp_forall( i = 0; i < N; i++; [affinity] )
    do { /* Do something */}
```

If affinity is type int, then “do statement” executed when

```
affinity % THREADS == MYTHREAD
```

If affinity is pointer, then “do statement” executed when

```
upc_threadof(affinity) == MYTHREAD
```

Memory Consistency

- Memory consistency can become an issue when data is shared among several threads
- `#include <upc_strict.h>`
  - Shared data is synchronized each time before access
- `#include <upc_relaxed.h>`
  - Threads are free to access shared data any time
  - Allows the compiler to optimize the order of shared access for the best performance
  - This is the default mode

Vector Addition

- Note that number of threads must be specified at compile time rather

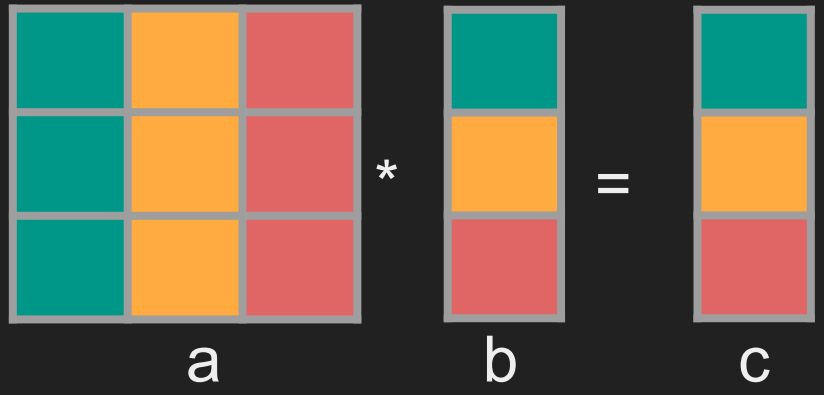
```
abed@ubuntu:~/Desktop/upc$ upc vect_add.upc -fupc-threads=4 -o vect_add
abed@ubuntu:~/Desktop/upc$ ./vect_add
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 5
8 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 1
10 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150
152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192
 194 196 198
```

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

#define N 100

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

void initVectors();
int printResult();

int main (void) {
    initArrays();
    upc_barrier;

    upc_forall(int i = 0 ; i < N; i++; i){
        vResult[i] = v1[i] + v2[i];
    }

    upc_barrier;
    printResult();

    return 0;
}
```

Matrix-Vector multiplication

```c
// matrix_vector_mult.upc
#include <stdio.h>
#include <upc.h>

#define ROWS 3

shared int a[ROWS][THREADS];
shared int b[THREADS];
shared int c[THREADS];

void initArrays();
int printResult();

int main (void) {
    initArrays();
    upc_barrier;

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

    upc_barrier;

    if (MYTHREAD == 0)
        printResult();

    return 0;
}
```

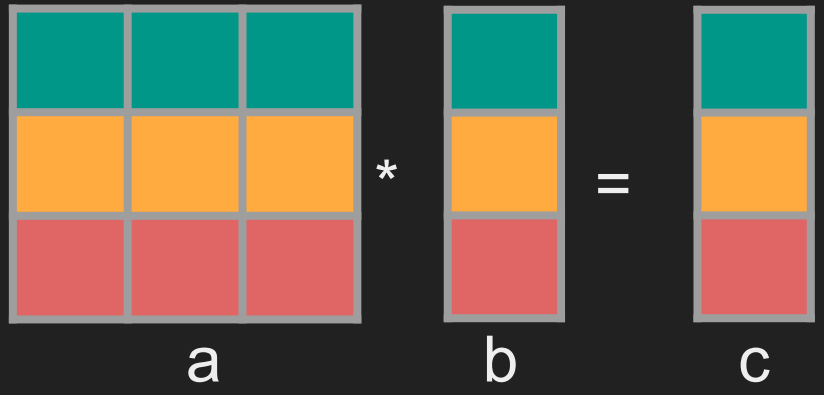
Thread 0
Thread 1
Thread 2



Matrix-Vector multiplication (better distribution)

```c
// matrix_vector_mult_blocks.upc
#include <stdio.h>
#include <upc.h>

#define ROWS 3

// This will distribute each row to its own thread
// (block oriented) rather by columns.
shared [THREADS] int a[ROWS][THREADS];
shared int b[THREADS];
shared int c[THREADS];

void initArrays();
int printResult();

int main (void) {
    initArrays();
    upc_barrier;

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

    upc_barrier;

    if (MYTHREAD == 0)
        printResult();

    return 0;
}
```

Thread 0
Thread 1
Thread 2



Collectives

- `#include <upc_collective.h>`
- Relocalization:
  - `upc_all_broadcast`
  - `upc_all_scatter`
  - `upc_all_gather`
  - `upc_all_gather_all`
  - `upc_all_exchange`
  - `upc_all_permute`
- Computational
  - `upc_all_reduce`
  - `upc_all_prefix_reduce`
  - `upc_all_sort`
- UPC Collective Operations Specifications V1.0

UPC VS MPI

UPC	MPI
Uses global address space Best for Random Access, Big data analysis, All-to-All simulations Advantages: Convenient to use Sharable data structure Closer to serial code Low overhead of communication Disadvantages: No locality control Does not scale Race condition	Uses message passing Best for massive independent jobs and simulation. Advantages: Scalability Locality Control All Communication is very explicit Disadvantages: Poor data structure Tedious pack/unpack code Data sharing scheme is poor Communication overhead gets high

DEMO

Questions?

THANKS