**Chandubhai S. Patel Institute of Technology-Changa**

**Department of Information Technology**

Subject Code: IT409

Subject Name: High Performance Computing Architecture

Semester: 7th /B.Tech

Academic Year: 2018-2019

Lab Manual

1. **Introduction to UPC**

This chapter provides a quick overview of Unified Parallel C (UPC). Discussions include a brief introduction to the roots of UPC, the UPC memory and programming models, compilers and run time environments. The language features of UPC will be examined in chapters 2 to 6.

**What is UPC?**

* UPC is a parallel extension of ANSI C.
* Adds parallelism to C
* Ability to read and write parallel memory with ease.
* Open source Implementation.

**UPC Programming Model**

* Distributed shared memory programming model.
* Use of data locality.
* Multithreading.
* Blocking and non-blocking function calls

**The UPC Memory model**

* Memory view is divided into private and shared spaces. Each thread has its own private space.

**Other Features of UPC**

* Data Distribution and Coherency
* Collective Operations
* Parallel I/O

Prerequisites:

* Good Knowledge of C Language (array and pointers).
* Uniform memory access Vs. Non-Uniform Memory Access.
* GCC Optimization levels.
* Thread and Multithreading.

1. **Programming in UPC**

Sample Program 1

#include <upc\_relaxed.h>

#include <stdio.h>

void main(){

printf(“Hello World from THREAD %d (of %d THREADS)\n”,MYTHREAD, THREADS);

}

Sample Program 2

#include <upc\_relaxed.h>

#define N 100\*THREADS

shared int v1[N], v2[N], v1plusv2[N];

void main(){

int i;

for(i= MYTHREAD; i<N; i+=THREADS)

v1plusv2[i]=v1[i]+v2[i];

}

Sample Program 3

#include <upc\_relaxed.h>

#include <stdio.h>

Void main(){

upc\_forall(i=0; i<N; i++; i){

printf("THREAD %d (of %d THREADS) performing iteration %d\n", MYTHREAD, THREADS, i);

}

}

**upc\_relaxed.h:** UPC Header file with memory consistency relaxed mode

**THREADS:** signifies the number of threads that the current execution is utilizing.

**MYTHREAD:** used to determine the thread number currently being executed

**upc\_forall:** (expression; expression; expression; affinity). This fourth parameter, alsoknown as the affinity field, accepts either an integer which is translated to (integer % THREADS); or an address which is used to determine the thread to which the address has its affinity.

**Shared:** To use the shared memory space

Few Examples of shared:

int local\_counter; // private variable

shared int global\_counter; // shared variable

shared int array1[N]; // shared array

shared [N/THREADS] int array2[N]; // shared array shared [ ] int array3[N]; // shared array

shared int \*ptr\_a; // private pointer to shared shared int \*shared ptr\_c; // shared pointer to shared

**upc\_barrier:** used to synchronize all threads before any of the threads continue. This is

generally used when a data dependency occurs between the threads.

**upc\_lock/upc\_unlock:** To ensure that the shared element is not accessed by other

threads while it is being updated by one thread is to use lock statements.

Guess Output:

#include <upc\_relaxed.h>

#include <stdio.h>

void main(){

if (MYTHREAD==0){

printf("Rcv’d: ‘Starting Execution’ from THREAD %d\n",MYTHREAD );

}

printf("Hello World from THREAD %d (of %d THREADS)\n",MYTHREAD, THREADS);

}

Guess Output:

#include <upc\_relaxed.h>

#include <stdio.h>

shared int a=0;

int b;

int computation(int temp){

return temp+5;

}

int main(){

int result=0, i=0;

do {

if (MYTHREAD==0){

result = computation(a);

a = result\*THREADS;

}

upc\_barrier;

b=a;

printf("THREAD %d: b = %d\n", MYTHREAD, b); i++;

} while (i<4); return 0;

}

Guess Output:

#include <upc\_relaxed.h>

#include <stdio.h>

#include <math.h>

#define N 1000

shared [] int arr[THREADS];

upc\_lock\_t \*lock;

int main (){

int i=0;

int index;

srand(MYTHREAD);

if ((lock=upc\_all\_lock\_alloc())==NULL)

upc\_global\_exit(1);

upc\_forall( i=0; i<N; i++; i){

index = rand()%THREADS;

upc\_lock(lock);

arr[index]+=1;

upc\_unlock(lock);

}

upc\_barrier;

if( MYTHREAD==0 ) {

for(i=0; i<THREADS; i++)

printf("TH%2d: # of arr is %d\n",i,arr[i]); upc\_lock\_free(lock);

return 0;

}

Why Lock is required in this program? What would be output of the program is lock is not applied at ant place?

**Memory Consistency**

The first method is to enforce strict data consistency, where shared data is synchronized each time before access. This implies that if the shared data is currently being updated by another thread, the thread will wait for a synchronization point before accessing the data. Strict mode also prevents the compiler from rearranging the sequence of indendent shared access operations for optimizations. This can result in significant overhead and should be avoided if possible.

The second mode is the relaxed mode, where the threads are free to access the shared data any time. This mode is the default mode because it allows the compiler to freely optimize the code to achieve better performance.

**Compiling and Running UPC Programs**

To compile and run your UPC program, it is best to refer to the compiler manual of your specific machine. In general you would compile using a UPC compiler, which takes a number of options; one of them can be the number of threads. In general, to compile a UPC code most compilers adopt similar compile time parameters:

<UPC compile command> <thread options> <optimizations> <code> –o <output>

for example:

upc –fthreads 4 -02 helloworld.c –o helloworld

Here upc is the compile command, -fthreads is the thread option, -02 specifies the optimization level desired, helloworld.c tells the compiler the program name ( most compilers support either .c or .upc extensions), and finally –o helloworld specifies the executable’s output name.

1. **Data and Pointers in UPC**

shared int x[12];

int y;

shared int z;

Assume 5 threads are available.

**Shared Data:** Distributes elementsx[0],x[1],..x[4]across threads 0, 1,..,4, respectivelyand so on…

Z will be allocated on one thread, thread 0.

**Private Data:** Since y is defined as a scalar private variable in line 2, UPC will allocatememory for the variable y on all the available threads.

**Blocking of Shared Arrays**

shared [block-size] array [number-of-elements];

Example

shared[3] int x[12];

Assuming the number of threads is 3, then elements x[0], x[1], and x[2] will have affinity to thread 0; x[3], x[4], and, x[5] will have affinity to thread 1, and so on...

When the [block-size] is omitted, the default block size of 1 is assumed.

How shared [2] int A[4][2] will be distributed?

Matrix by Vector Multiply

#include<upc\_relaxed.h>

#define N 100\*THREADS

shared [N] double A[N][N];

shared double b[N], x[N];

void main(){

int i,j;

* reading the elements of matrix A and the vector x and initializing the vector b to zeros upc\_forall(i=0;i<N;i++;i)

for(j=0;j<N;j++) b[i]+=A[i][j]\*x[j] ;

}

**UPC Shared and Private Pointers**

There are four distinct possibilities:

1. private pointers pointing to the private space(int \*p1)
2. private pointers pointing to the shared space(shared int \*p2)
3. shared pointers pointing to the shared space(int \*shared p3)
4. shared pointers pointing to the private space(shared int \*shared p4)

**Shared and Private Pointer address format**

Unlike ordinary C pointers, a UPC pointer-to-shared has to keep track of a number of things. These are the thread number; the virtual address of the block, and the phase that indicates to which item in the block the pointer is pointing.

**Special UPC Pointer Functions and Operators**

**upc\_threadof (shared void \*ptr):** Returns the number of the thread that has affinity tothe shared object pointed to by ptr.

**upc\_phaseof (shared void \*ptr):** Returns the position within the block of the pointerptr. **upc\_addrfield(shared void \*ptr):** Returns the address of the bloc which is pointed at bythe pointer-to-shared ptr.

**upc\_localsizeof(type-name or expression):** Returns the size of the local portion of ashared object.

**upc\_blocksizeof (type-name or expression):** Returns the block\_size associated withoperand.

**upc\_elemsizeof (type-name or expression):** Returns the size (in bytes) of the left-mosttype that is not an array.

**Casting of Shared to Private Pointers**

shared int x[THREADS];

int \*p;

p=(int \*)&x[MYTHREAD];

The Role of Casting of Shared Pointers

shared [3] int \*p;

shared [5] int \*q;

p=q;

**UPC Pointer Arithmetic**

Assuming we have 4 threads

#define N 16

shared int x[N];

shared int \*p=&x[5];

p=p+3;

How Shared Pointer Arithmetic Follows Blocking.

#define N 16

shared [3] int x[N];

shared int \*p=&x[4];

p=p+7;

**UPC String Handling Functions**

upc\_memcpy( dst, src, n)

upc\_memput( dst, src, n)

upc\_memget( dst, src, n)

copies shared to shared memory.

copies from private to shared memory.

copies from shared to private memory.

Example(Assuming that SIZE%THREADS == 0)

#include <upc\_relaxed.h>

#define SIZE 16000

shared int data[SIZE];

shared [] int th0\_data[SIZE];

int main(){

int i, sum;

sum = 0;

if (MYTHREAD==0){

for( i=0; i<THREADS; i++ )

upc\_memcpy(&th0\_data[i\*(SIZE/THREADS)], &data[i],

(SIZE/THREADS)\*sizeof(int));

for( i=0; i<SIZE; i++ )

sum += th0\_data[i];

}

return 0;

}

Example

#include <upc\_relaxed.h>

#define SIZE 16000

shared int data[SIZE];

int main(){

int i, sum;

int localbuf[SIZE/THREADS];

sum = MYTHREAD;

for( i=0; i<SIZE/THREADS; i++ ) {

localbuf[i] = sum;

sum += i\*THREADS;

}

upc\_memput(&data[MYTHREAD],localbuf, (SIZE/THREADS)\*sizeof(int)); return 0;

}

Example:

#include <upc\_relaxed.h>

#define SIZE 10000

shared int data[SIZE];

int main(){

int i, sum;

int localbuf[SIZE];

for( i=0; i<THREADS; i++ )

upc\_memget(&localbuf[i\*(SIZE/THREADS)],&data[MYTHREAD], (SIZE/THREADS)\* sizeof(int));

sum = 0;

for( i=0; i<SIZE; i++ )

sum += localbuf[i];

return 0;}

1. **Work Distribution**

**Data Distribution for Work Sharing**

#include <upc\_relaxed.h>

#define IMG\_SIZE 512

shared [(IMG\_SIZE\*IMG\_SIZE)/THREADS] int image[IMG\_SIZE][IMG\_SIZE];

image[0][0]

...

image[(img\_size/threads)-1][img\_size-1]

**Thread0**

image[(img\_size/threads)][0]

…

image[2\*(img\_size/threads)-1][img\_size-1]

**Thread1**

Matrix by Vector Multiply

#include<upc\_relaxed.h>

#define N 100\*THREADS

shared [N] double A[N][N];

shared double b[N], x[N];

void main()

{

int i,j;

* reading the elements of matrix A and the vector x and initializing the vector b to zeros upc\_forall(i=0;i<N;i++;i)

for(j=0;j<N;j++)

b[i]+=A[i][j]\*x[j] ;

}

1. **Synchronization and MemoryConsistency**

**Barrier Synchronization**

UPC provides two basic kinds of barriers, blocking barriers and split-phase barriers (non-blocking). The blocking barrier is invoked by calling the function upc\_barrier and the split-phase barrier is obtained by calling the function pair upc\_notify and upc\_wait.

Example: Barrier Synchronization

shared [N]int A[N][N];

shared [N]int C[N][N];

shared [N]int B[N][N];

shared [N]int ACsum[N][N];

shared [N]int Bsqr[N][N];

shared [N]int Result[N][N];

void matrix\_multiplication (shared[N] int result[N][N],shared[N] int m1[N][N], shared[N] int m2[N][N]){

int i, j, l, sum;

upc\_forall(i=0;i<N;i++; &m1[i][0]){

for(j=0;j<N;j++){

sum=0;

for(l=0;l<N;l++)

sum+=m1[i][l]\*m2[l][j];

result [i][j]=sum;

}

}

}

matrix\_multiplication(Bsqr,B,B);

upc\_notify 1;

upc\_forall(i=0;i<N;i++;&A[i][0]){

for(j=0;j<N;j++)

ACsum[i][j]+=A[i][j]+C[i][j];

}

upc\_wait 1;

matrix\_multiplication(Result, ACsum, Bsqr);

**Synchronization Locks**

void upc\_lock(upc\_lock\_t \*l): allow locking

void upc\_unlock(upc\_lock\_t \*l): allow unlocking

int upc\_lock\_attempt(upc\_lock\_t \*l): returns 1 on successful locking and 0 otherwise. This may improve performance by avoiding busy waits when a lock is not available.

Example: Numerical Integration with Locks

//Numerical Integration

//Example - The Famous PI

#include<upc\_relaxed.h>

#include<math.h>

#define N 1000000

#define f(x) (1.0/(1.0+x\*x))

upc\_lock\_t \*l;

shared float pi = 0.0;

void main(void){

float local\_pi=0.0;

int i;

l=upc\_all\_lock\_alloc();

upc\_forall(i=0;i<N;i++; i)

local\_pi +=(float) f((.5+i)/(N));

local\_pi \*= (float) (4.0 / N);

upc\_lock(l);

pi += local\_pi;

upc\_unlock(l);

upc\_barrier; // Ensure all is done

if(MYTHREAD==0) printf("PI=%f\n",pi);

if(MYTHREAD==0) upc\_lock\_free(l);

}

Example: More Locks

//create the locks

upc\_lock\_t \*l1;

upc\_lock\_t \*l2;

shared float v1=1.0, v2=2.0;

l1=upc\_all\_lock\_alloc();

l2=upc\_all\_lock\_alloc();

if (MYTHREAD>THREADS/2) update\_v1();

else update\_v2();

void update\_v1()

{

upc\_lock(l1);

v1=expression1(v1);

upc\_unlock(l1);

}

void update\_v2()

{

upc\_lock(l2);

v2=expression2(v2);

upc\_unlock(l2);

}

1. **Dynamic Memory Allocation in UPC**

Dynamic shared memory allocation in UPC can be collective or non-collective, global or local.There are four allocation functions:

upc\_all\_alloc

upc\_global\_alloc

upc\_alloc

upc\_local\_alloc (deprecated in language specification V1.1)

The first two are similar:

shared void \*upc\_all\_alloc (size\_t nblocks, size\_t nbytes)

shared void \*upc\_global\_alloc (size\_t nblocks, size\_t nbytes)

Both functions are *global*: they allocate shared space across all threads, compatible with the declaration:

shared [nbytes] char[nblocks\*nbytes] upc\_all\_alloc is a collective function; that is, it must be called by all threads with the same arguments, and it returns the same pointer value on all threads. On the other hand, upc\_global\_alloc is not a collective function; it is called by one thread. If called by more than one thread, multiple regions are allocated, and each calling thread gets a pointer to its own allocation.SS

The other two functions are *local*: they allocate shared memory that has affinity to the calling thread only. The function upc\_local\_alloc is in the original language specification, but is deprecated in specification V1.1, to be replaced by upc\_alloc. Both functions have the same effect: upc\_alloc allocates nbytes bytes, while upc\_local\_alloc allocates nblocks\*nbytes

bytes. The functions are similar to the declaration: shared [] char [nbytes] except that this allocates shared memory with affinity to thread 0 only.

**Freeing Memory**

The function void upc\_free (shared void \*ptr) frees the dynamically allocated shared storage pointed to by ptr. In the case of memory allocated by upc\_all\_alloc, any thread, may call upc\_free to free the memory; but only one such call has any effect. Example shows upc\_free is used to free memory allocated by the upc\_all\_alloc function.

Example: Shared Memory Dynamic Allocation

shared [10] int \*table;

int main(){

* allocate a buffer of 10\*THREADS, with block\_size of 10 elements table = (shared [10] int \*)upc\_all\_alloc(THREADS, 10\* sizeof(int)); /\* do some work here \*/

(…)

/\* free the table, any thread may free table \*/ if (MYTHREAD==0)

upc\_free( table );

}

Example: Different scenarios of Shared Memory Dynamic Allocations

/\* shared variable declarations \*/

shared [5] int \*p1, \*p2, \*p3;

shared [5] int \* shared p4, \* shared p5;

/\* Allocate 25 elements per thread, with each thread doing its portion of the allocation. –

COLLECTIVE CALL \*/

p1 = (shared [5] int \*)upc\_all\_alloc(5\*THREADS, 5\*sizeof(int));

/\* Allocate 25 elements per thread, but just run the allocation on thread 5. – NON

COLLECTIVE CALL \*/

if (MYTHREAD == 5)

p2 = (shared [5] int \*)upc\_global\_alloc(5\*THREADS, 5\*sizeof(int)); /\* Allocate 5 elements only on thread 3. NON COLLECTIVE CALL \*/ if (MYTHREAD == 3)

p3 = (shared [5] int \*)upc\_alloc(sizeof(int)\*5);

/\* Allocate 25 elements per thread, just run the allocation on thread 4, but have the result be visible everywhere.- NONCOLLECTIVE CALL \*/ if (MYTHREAD == 4)

p4 = (shared [5] int shared \*)upc\_global\_alloc(5\*THREADS, 5\*sizeof(int)); /\* Allocate 5 elements only on thread 2, but have the result visible on all threads. \*/ if (MYTHREAD == 2)

p5 = (shared [5] int shared \*)upc\_alloc(sizeof(int)\*5); /\* De-allocate p1, any thread may free p1\*/ if( MYTHREAD == 0 )

upc\_free( p1 );

/\* De-allocate p2, only thread 5 may free p2\*/ if( MYTHREAD == 5 )

upc\_free( p2 );

/\* De-allocate p3\*/

if( MYTHREAD == 3 )

upc\_free( p3 );

/\* De-allocate p4 & p5 \*/

if( MYTHREAD == 0 ) {

upc\_free( p4 );

upc\_free( p5 );

}

1. **UPC Optimization**

There are several ways to enhance the performance of UPC through either compiler and runtime optimizations and/or hand–tuning. These are discussed in this chapter along with specific examples.

**How to Exploit the Opportunities for Performance Enhancement**

Performance optimizations are typically possible through:

* Compiler optimizations
* Run-time system
* Hand tuning

**Compiler and Runtime Optimizations**

An advanced programmer should become familiar with the UPC specific compiler optimization options. The user should also be aware of whether the vendor has a run time system that can help optimize your code and how to set respective environment variables. When everything else fails, the following hand tuning techniques can be used.

**List of Hand Tunings for UPC Code Optimization**

The performance of UPC code can be improved using the following hand tuning methods:

1. Use local pointers instead of shared pointers when dealing with local shared data, through casting and assignments
2. Use block copy instead of copying elements one by one with a loop
3. Overlap remote accesses with local processing using split-phase barriers

*Using local pointers instead of shared pointers*

UPC compilers may generate code which takes longer to access local shared data than private data. Thus, for better performance all UPC local shared accesses must be turned into UPC private accesses. This step is called privatization. Example illustrates how to privatize local shared accesses in a UPC code, or in other words, how to convert UPC local shared accesses to UPC private accesses to obtain an effective memory bandwidth.

Example Privatization example

int \*pa, \*pc;

upc\_forall(i=0;i<N;i++;&A[i][0]) {

\*pa = (int\*) &A[i][0];

\*pc = (int\*) &C[i][0];

for(j=0;j<P;j++)

pa[j]+=pc[j];

}

Pointer arithmetic is typically faster using private pointers than shared pointers. In some cases, pointer de-referencing can be an order of magnitude faster.

*Aggregation of Accesses Using Block Copy*

When UPC shared remote accesses are needed, aggregating such accesses and fetching them as a block is better than multiple reads/writes since latency and other overheads only appear once. Example shows how to use a block copy instead of a

standard single element copy. The block copy is done using a string function, very similar to the ones found in the C language.

Example Block copy by string function copy example

Instead of

shared [] int a[1000], b[1000];

* Copy element by element for (j=0; j<1000; j++) b[j]=a[j];
* Copy the whole array at once using string functions upc\_memcpy(b, a , sizeof(a));

*Overlapping Remote Accesses with Local Processing*

In order to hide the time spent in remote shared accesses, it is possible to overlap communication with computations. This is can be done using split-phase barriers instead of blocking barriers. In this case local processing can be done while waiting for data or synchronization. The example shows a brief implementation of computation and communication overlapping.

Ghost zones are prefetched and while waiting for this prefetching, the computation can be done on all the local shared data except for the ghost zones. After completion of such processing, upc\_wait() waits for all threads to complete the communication step, and thereafter the ghost zone can be processed.

Example: Overlapping and split-phase barriers

upc\_memcpy(ghost\_copy, ghost\_zone, size);

upc\_notify;

* work on everything but the ghost\_zones upc\_wait;
* work with the ghost\_zones

1. **Edge Detection in C**

#define BYTE unsigned char

BYTE orig[N][N],edge[N][N];

int Sobel(){

int i,j,d1,d2;

double magnitude;

for (i=1; i<N-1; i++){

for (j=1; j<N-1; j++){

d1 = (int) orig[i-1][j+1] - orig[i-1][j-1];

d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;

d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];

d2 = (int) orig[i-1][j-1] - orig[i+1][j-1];

d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;

d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];

magnitude = sqrt(d1\*d1+d2\*d2);

edge[i][j]= magnitude>255? 255:(BYTE)magnitude;

}

}

return 0;

}

Edge Detection could be written in UPC .

#define BYTE unsigned char

shared [N\*N/THREADS] BYTE orig[N][N],edge[N][N]; int Sobel(){

int i,j,d1,d2;

double magnitude;

upc\_forall (i=1; i<N-1; i++; &edge[i][0]){

for (j=1; j<N-1; j++){

d1 = (int) orig[i-1][j+1] - orig[i-1][j-1];

d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;

d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];

d2 = (int) orig[i-1][j-1] - orig[i+1][j-1];

d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;

d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];

magnitude = sqrt(d1\*d1+d2\*d2);

edge[i][j] = magnitude>255? 255 : (BYTE)magnitude;

}

}

return 0;

}

1. **N-Queens**

**Nqueens in UPC**

//Main program - variables

shared int number\_solns[THREADS];

// parameters

shared int n; // Problem size

shared int l; // Distribution

shared int method; // Round-robin / chunking

//Main program - initialization

if (MYTHREAD==0) {

n=atoi(argv[1]);

if ((n<=0) || (n>16)) {

fprintf(stderr,"0<n<17\n");

upc\_global\_exit(0);

}

l=atoi(argv[2]);

if ((l<0) || (l>=n)) {

fprintf(stderr,"0<=l<n\n");

upc\_global\_exit(0);

}

}

...

//Main program - execution

upc\_barrier; // make sure thread 0 has set the parameters number\_solns[MYTHREAD] = sched(n,l,method); upc\_barrier; // Complete all solutions before reduction nsols=0;

if (MYTHREAD==0) {

for(i=0;i<THREADS;i++)

nsols+=number\_solns[i];

}

...

//Code for job distribution

int sched(int n, int l, int method) {

* Distribution in a round-robin fashion: if (method==roundrobin)

upc\_forall(j=0;j<njobs;j++; j) { call sequential algorithm

}

if (method==chunk)

* or Distribution in a chunking fashion:

upc\_forall(j=0;j<njobs;j++; (j\*THREADS)/njobs ) {

call sequential algorithm

}

}