Parallel Programming Models and Unified Parallel C

Tarek El-Ghazawi

Introduction

- UPC – Unified Parallel C
- Set of specs for a parallel C
  - v1.0 completed February of 2001
  - v1.1.1 in October of 2003
  - v1.2 in May of 2005
- Compiler implementations by vendors and others
- Consortium of government, academia, and HPC vendors including IDA CCS, GWU, UCB, MTU, UMN, ARSC, UMCP, U of florida, ANL, LBNL, LLNL, DoD, DoE, HP, Cray, IBM, Sun, Intrepid, Etnus, …
Introductions cont.

• UPC compilers are now available for most HPC platforms and clusters
  - Some are open source
• A debugger is available and a performance analysis tool is in the works
• Benchmarks, programming examples, and compiler testing suite(s) are available
• Visit www.upcworld.org or upc.gwu.edu for more information

Parallel Programming Models

• What is a programming model?
  - An abstract virtual machine
  - A view of data and execution
  - The logical interface between architecture and applications
• Why Programming Models?
  - Decouple applications and architectures
    - Write applications that run effectively across architectures
    - Design new architectures that can effectively support legacy applications
• Programming Model Design Considerations
  - Expose modern architectural features to exploit machine power and improve performance
  - Maintain Ease of Use
Programming Models

- Common Parallel Programming models
  - Data Parallel
  - Message Passing
  - Shared Memory
  - Distributed Shared Memory
  - ...
- Hybrid models
  - Shared Memory under Message Passing
  - ...

![Diagram of Programming Models]

- MPI
- OpenMP
- UPC
- DSM/PGAS
- Process/Thread
- Address Space
The Partitioned Global Address Space (PGAS) Model

- Aka the DSM model
- Concurrent threads with a partitioned shared space
  - Similar to the shared memory
  - Memory partition $M_i$ has affinity to thread $T_h_i$

(+ive):
- Helps exploiting locality
- Simple statements as SM

(-ive):
- Synchronization

• UPC, also CAF and Titanium

What is UPC?

- Unified Parallel C
- An explicit parallel extension of ISO C
- A distributed shared memory parallel programming language
UPC Execution Model

- A number of threads working independently in a **SPMD** fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
  - Number of threads specified at compile-time or run-time
- Synchronization when needed
  - Barriers
  - Locks
  - Memory consistency control

UPC Memory Model

- A pointer-to-shared can reference all locations in the shared space, but there is data-thread affinity
- A private pointer may reference addresses in its private space or its local portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory
A collection of threads operating in a single global address space, which is logically partitioned among threads. Each thread has affinity with a portion of the globally shared address space. Each thread has also a private space.

A First Example: Vector addition

```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD==i%THREADS)
            v1plusv2[i]=v1[i]+v2[i];
}
```
2nd Example: A More Efficient Implementation

//vect_add.c
#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];
}

3rd Example: A More Convenient Implementation with upc_forall

//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], v1plusv2[N];
void main()
{
    int i;
    upc_forall(i=0; i<N; i++; i)
        v1plusv2[i]=v1[i]+v2[i];
}
Example: UPC Matrix-Vector Multiplication- Default Distribution

// vect_mat_mult.c
#include <upc_relaxed.h>

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

Data Distribution

\[
\begin{array}{ccc}
\text{Th. 0} & \text{Th. 1} & \text{Th. 2} \\
\times & \text{Thread 0} & \text{Thread 1} & \text{Thread 2} \\
\text{C} & \text{A} & \text{B} \\
\end{array}
\]
A Better Data Distribution

Example: UPC Matrix-Vector Multiplication - The Better Distribution

```c
// vect_mat_mult.c
#include <upc_relaxed.h>

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

void main (void) {
    int i, j;
    upc_forall( i = 0 ; i < THREADS ; i++; i) {
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```
Examples of Shared and Private Data

**Layout:**
Assume THREADS = 3

shared int x; /*x will have affinity to thread 0*/
shared int y[THREADS];
int z;

will result in the layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[0]</td>
<td>y[1]</td>
<td>y[2]</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
</tbody>
</table>

shared int A[4][THREADS];

will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td>A[0][2]</td>
</tr>
</tbody>
</table>
shared int A[2][2*THREADS];
will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>...</th>
<th>Thread (THREADS-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td></td>
<td>A[0][THREADS-1]</td>
</tr>
<tr>
<td>A[0][THREADS]</td>
<td>A[0][THREADS+1]</td>
<td></td>
<td>A[0][2*THREADS-1]</td>
</tr>
</tbody>
</table>

Blocking of Shared Arrays

- Default block size is 1
- Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.
- A block size is specified in the declaration as follows:
  - shared [block-size] type array[N];
  - e.g.: shared [4] int a[16];
Blocking of Shared Arrays

• Block size and THREADS determine affinity
• The term affinity means in which thread’s local shared-memory space, a shared data item will reside
• Element $i$ of a blocked array has affinity to thread:
  \[
  \left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS}
  \]

Shared and Private Data

• Shared objects placed in memory based on affinity
• Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer
• All non-array shared qualified objects, i.e. shared scalars, have affinity to thread 0
• Threads access shared and private data
Assume THREADS = 4
will result in the following data layout:

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[3][0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[3][1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[3][2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special Operators

- **upc.localsizeof(type-name or expression);**
  returns the size of the local portion of a shared object
- **upc.blocksizeof(type-name or expression);**
  returns the blocking factor associated with the argument
- **upc.elemsizeof(type-name or expression);**
  returns the size (in bytes) of the left-most type that is not an array
Usage Example of Special Operators

typedef shared int sharray[10*THREADS];
sharray a;
char i;

• upc.localsizeof(sharray)  \rightarrow  10*\text{sizeof(int)}
• upc.localsizeof(a)  \rightarrow  10*\text{sizeof(int)}
• upc.localsizeof(i)  \rightarrow  1
• upc.blocksizeof(a)  \rightarrow  1
• upc.elementsizeof(a)  \rightarrow  \text{sizeof(int)}

String functions in UPC

• UPC provides standard library functions to move data to/from shared memory
• Can be used to move chunks in the shared space or between shared and private spaces
String functions in UPC

- **Equivalent of memcpy:**
  - `upc_memcpy(dst, src, size)`
    - copy from shared to shared
  - `upc_memput(dst, src, size)`
    - copy from private to shared
  - `upc_memget(dst, src, size)`
    - copy from shared to private

- **Equivalent of memset:**
  - `upc_memset(dst, char, size)`
    - initializes shared memory with a character

- The shared block must be a contiguous with all of its elements having the same affinity

UPC Pointers

Where does it point to?

<table>
<thead>
<tr>
<th></th>
<th>Private</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>PP</td>
<td>PS</td>
</tr>
<tr>
<td>Shared</td>
<td>SP</td>
<td>SS</td>
</tr>
</tbody>
</table>

Where does it reside?
UPC Pointers

• How to declare them?
  - `int *p1; /* private pointer pointing locally */`
  - `shared int *p2; /* private pointer pointing into the shared space */`
  - `int *shared p3; /* shared pointer pointing locally */`
  - `shared int *shared p4; /* shared pointer pointing into the shared space */`

• You may find many using “shared pointer” to mean a pointer pointing to a shared object, e.g. equivalent to p2 but could be p4 as well.


• **What are the common usages?**
  
  - `int *p1; /* access to private data or to local shared data */`
  
  - `shared int *p2; /* independent access of threads to data in shared space */`
  
  - `int *shared p3; /* not recommended*/`
  
  - `shared int *shared p4; /* common access of all threads to data in the shared space*/`

---

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

<table>
<thead>
<tr>
<th>Thread #</th>
<th>Block Address</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>49</td>
<td>48</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>0</td>
<td>38</td>
<td>37</td>
</tr>
</tbody>
</table>

• **Example: Cray T3E implementation**
UPC Pointers

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa!
- When casting a pointer-to-shared to a private pointer, 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

Special Functions

- size_t upc_threadof(shared void *ptr);
  returns the thread number that has affinity to the object pointed to by ptr
- size_t upc_phaseof(shared void *ptr);
  returns the index (position within the block) of the object which is pointed to by ptr
- size_t upc_addrfield(shared void *ptr);
  returns the address of the block which is pointed at by the pointer to shared
- shared void *upc_resetphase(shared void *ptr);
  resets the phase to zero
- size_t upc_affinitysize(size_t ntotal, size_t nbytes, size_t thr);
  returns the exact size of the local portion of the data in a shared object with affinity to a given thread
pointer to shared Arithmetic Examples:
Assume THREADS = 4
#define N 16
shared int x[N];
shared int *dp=&x[5], *dp1;
dp1 = dp + 9;
Assume THREADS = 4

```c
shared[3] int x[N], *dp=&x[5], *dp1;

dp1 = dp + 9;
```
Example Pointer Castings and Mismatched Assignments:

• Pointer Casting

```c
shared int x[THREADS];
int *p;
p = (int *) &x[MYTHREAD]; /* p points to x[MYTHREAD] */
```

- Each of the private pointers will point at the x element which has affinity with its thread, i.e. MYTHREAD

• Mismatched Assignments

Assume THREADS = 4

```c
shared int x[N];
shared[3] int *dp=&x[5], *dp1;
dp1 = dp + 9;
```

- The last statement assigns to dp1 a value that is 9 positions beyond dp
- The pointer will follow its own blocking and not that of the array
UPC Pointers

- Given the declarations
  \[
  \text{shared[3] int } *p; \\
  \text{shared[5] int } *q; \\
  \]
- Then
  \[
  p=q; /* is acceptable (an implementation may require an explicit cast, e.g. \(p=(\text{shared[3]}q);\) */ \\
  \]
- Pointer \(p\), however, will follow pointer arithmetic for blocks of 3, not 5 !!
- A pointer cast sets the phase to 0
Worksharing with upc_forall

- Distributes independent iteration across threads in the way you wish—typically used to boost locality exploitation in a convenient way
- Simple C-like syntax and semantics
  \[
  \text{upc_forall}(\text{init}; \text{test}; \text{loop}; \text{affinity})
  \]
  
  statement
  - Affinity could be an integer expression, or a
  - Reference to (address of) a shared object

---

Work Sharing and Exploiting Locality via upc_forall()

- Example 1: explicit affinity using shared references
  
  \[
  \text{shared int } a[100], b[100], c[100]; \\
  \text{int } i; \\
  \text{upc_forall (i}=0; \text{i}<100; \text{i++; &a[i])} \\
  \quad a[i] = b[i] \times c[i];
  \]

- Example 2: implicit affinity with integer expressions and distribution in a round-robin fashion
  
  \[
  \text{shared int } a[100], b[100], c[100]; \\
  \text{int } i; \\
  \text{upc_forall (i}=0; \text{i}<100; \text{i++; i) } \\
  \quad a[i] = b[i] \times c[i];
  \]

\textbf{Note:} Examples 1 and 2 result in the same distribution
Work Sharing: upc_forall()

- Example 3: Implicitly with distribution by chunks
  shared int a[100], b[100], c[100];
  int i;
  upc_forall (i=0; i<100; i++; (i*THREADS)/100)
    a[i] = b[i] * c[i];

- Assuming 4 threads, the following results

<table>
<thead>
<tr>
<th>i</th>
<th>i*THREADS</th>
<th>i*THREADS/100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..24</td>
<td>0.96</td>
<td>0</td>
</tr>
<tr>
<td>25..49</td>
<td>100.196</td>
<td>1</td>
</tr>
<tr>
<td>50..74</td>
<td>200.296</td>
<td>2</td>
</tr>
<tr>
<td>75..99</td>
<td>300.396</td>
<td>3</td>
</tr>
</tbody>
</table>

Distributing Multidimensional Data

- Uses the inherent contiguous memory layout of C multidimensional arrays
  shared [BLOCKSIZE] double grids[N][N];
  Distribution depends on the value of BLOCKSIZE,
  - Column Blocks BLOCKSIZE= N/THREADS
  - Row Block BLOCKSIZE=N
  - Distribution by
    
  - BLOCKSIZE=N*N
  - BLOCKSIZE = infinite
2D Heat Conduction Problem

Based on the 2D Partial Differential Equation (1), 2D Heat Conduction problem is similar to a 4-point stencil operation, as seen in (2):

\[
\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha} \frac{\partial T}{\partial t} \quad (1)
\]

\[
T_{i,j}^{t+1} = \frac{1}{4 \cdot \alpha} \left( T_{i-1,j}^t + T_{i+1,j}^t + T_{i,j-1}^t + T_{i,j+1}^t \right) \quad (2)
\]

Because of the time steps, Typically, two grids are used

```
shared [BLOCKSIZE] double grids[2][N][N];
shared double dTmax_local[THREADS];
int nr_iter,i,x,y,z,dg,sg,finished;
double dTmax, dT, T;
do {
    dTmax = 0.0;
    for( y=1; y<N-1; y++ )
    {
        upc_forall( x=1; x<N-1; x++; &grids[sg][y][x] )
        {
            T = (grids[sg][y-1][x] + grids[sg][y+1][x] + 
grids[sg][z][y][x-1] + grids[sg][z][y][x+1]) / 4.0;
            dT = T - grids[sg][y][x];
            grids[dg][y][x] = T;
            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }
    }
}
```

Work distribution, according to the defined BLOCKSIZE of grids[][]]

**HERE, generic expression, working for any BLOCKSIZE**

4-point Stencil
if( dTmax < epsilon )
    finished = 1;
else
{
    // swapping the source & destination "pointers"
    dg = sg;
    sg = !sg;
}
    nr_iter++;
} while( !finished );
upc_barrier;

Dynamic Memory Allocation in UPC

• Dynamic memory allocation of shared memory is available in UPC
• Functions can be collective or not
• A collective function has to be called by every thread and will return the same value to all of them
• As a convention, the name of a collective function typically includes “all”
Collective Global Memory Allocation

shared void *upc_all Alloc
(size_t nblocks, size_t nbytes);

nbblocks: number of blocks
nbytes: block size

- This function has the same result as upc_global_alloc. But this is a collective function, which is expected to be called by all threads
- All the threads will get the same pointer
- Equivalent to:
  shared [nbytes] char[nblocks * nbytes]

Collective Global Memory Allocation

Thread0  Thread1  ThreadTHREADS-1

shared [N] int *ptr;
ptr = (shared [N] int *)
upc_all_alloc( THREADS, N*sizeof( int ) );
#define CHECK_MEM(var) {
    if (var == NULL) {
        printf("TH%02d: ERROR: %s == NULL\n", MYTHREAD, #var);
        upc_global_exit(1);
    }
}

shared [BLOCKSIZE] double *sh_grids;

void heat_conduction(shared [BLOCKSIZE] double (*grids)[N][N]) {
    for (y=1; y<N-1; y++)
    {
        upc_forall( x=1; x<N-1; x++; &grids[sg][y][x] )
        {
            T = (grids[sg][y+1][x] + ...
            ...
        }
        while(finished == 0);
        return nr_iter;
    }
}

int main(void)
{
    int nr_iter;

    /* allocate the memory required for grids[2][N][N] */
    sh_grids = (shared [BLOCKSIZE] double *)
        upc_all_alloc( 2*N*N/BLOCKSIZE,
                       BLOCKSIZE*sizeof(double));
    CHECK_MEM(sh_grids);
    ...

    /* performs the heat conduction computations */
    nr_iter = heat_conduction(
        (shared [BLOCKSIZE] double (*)[N][N]) sh_grids);
    ...
}
shared void *upc_global_alloc
(size_t nblocks, size_t nbytes);

nblocks : number of blocks
nbytes : block size

• Non collective, expected to be called by one thread
• The calling thread allocates a contiguous memory region in the shared space
• Space allocated per calling thread is equivalent to :
  shared [nbytes] char[nblocks * nbytes]
• If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer

shared [N] int *ptr;

ptr =
  (shared [N] int *)
  upc_global_alloc( THREADS, N*sizeof( int ));

shared [N] int *shared
  myptr[THREADS];

myptr[MYTHREAD] =
  (shared [N] int *)
  upc_global_alloc( THREADS, N*sizeof( int ));
Local-Shared Memory Allocation

shared void *upc_alloc (size_t nbytes);

nbytes: block size

- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory region in the local-shared space of the calling thread
- Space allocated per calling thread is equivalent to:
  \[
  \text{shared}[] \text{char}[\text{nbytes}]
  \]
- If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer

```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof(int));
```
• Blocking can also be done by 2D “cells”, of equal size across THREADS

• Works best with \( N \) being a power of 2

---

**Determining \( \text{DIMX} \) and \( \text{DIMY} \):**

\[
\text{NO\_COLS} = \text{NO\_ROWS} = 1;
\]

```
for( i=2, j=0; i<=\text{THREADS}; i<<=1, j++ ) {
    if( (j\%3)==0 )
        \text{NO\_COLS} <<= 1;
    else if((j\%3)==1)
        \text{NO\_ROWS} <<= 1;
}
```

\[
\text{DIMX} = \frac{\text{N}}{\text{NO\_COLS}}; \\
\text{DIMY} = \frac{\text{N}}{\text{NO\_ROWS}};
\]
Accessing one element of those 3D shared cells (by a macro):

#define CELL_SIZE DIMY*DIMX
struct gridcell_s {
    double cell[CELL_SIZE];
};
typedef struct gridcell_s gridcell_t;

shared gridcell_t cell_grids[2][THREADS];

#define grids(gridno, y, x) \
    cell_grids[gridno][((y)/DIMY)*NO_COLS + ((x)/DIMX)].cell[ \
        ((y)%DIMY)*DIMX + ((x)%DIMX)]

2* One cell per thread

Linearization – 2D into a 1+1D (using a C Macro)

Which THREAD?

Which Offset in the cell?

---

2D Heat Conduction Example w 2D-Cells

typedef struct chunk_s chunk_t;
struct chunk_s {
    shared [] double *chunk;
};

int N;
shared chunk_t sh_grids[2][THREADS];
shared double dTmax_local[THREADS];

#define grids(no,y,x) sh_grids[no][((y)*N+(x))/(N*N*N/THREADS)].chunk \
    [((y)*N+(x))%(N*N*N/THREADS)]

int heat_conduction(shared chunk_t (*sh_grids)[THREADS]) {
    // grids[][][] has to be changed to grids(,,)
}

---
int main(int argc, char **argv)
{
    int nr_iter, no;
    // get N as parameter
    ...

    for( no=0; no<2; no++ ) /* allocate */
    {
        sh_grids[no][MYTHREAD].chunk =
        ~((shared []) double *) upc_alloc
        (N*N/THREADS*sizeof( double ));
        CHECK_MEM( sh_grids[no][MYTHREAD].chunk );
    }
    ...
    /* performs the heat conduction computation */
    nr_iter = heat_conduction(sh_grids);
    ...
}

Memory Space Clean-up

void upc_free(shared void *ptr);

- The upc_free function frees the dynamically allocated shared memory pointed to by ptr
- upc_free is not collective
Example: Matrix Multiplication in UPC

- Given two integer matrices \( A(N \times P) \) and \( B(P \times M) \), we want to compute \( C = A \times B \).
- Entries \( c_{ij} \) in \( C \) are computed by the formula:

\[
c_{ij} = \sum_{l=1}^{P} a_{il} \times b_{lj}
\]

Doing it in C

```c
#include <stdlib.h>

#define N 4
#define P 4
#define M 4

int a[N][P] = {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

void main (void) {
    int i, j, l;
    for (i = 0 ; i < N ; i++) {
        for (j = 0 ; j < M ; j++) {
            c[i][j] = 0;
            for (l = 0 ; l < P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
Domain Decomposition for UPC

Exploiting locality in matrix multiplication

- A \((N \times P)\) is decomposed row-wise into blocks of size \((N \times P) / \text{THREADS}\) as shown below:

- B\((P \times M)\) is decomposed column-wise into \(M / \text{THREADS}\) blocks as shown below:

\[
\begin{array}{c}
\text{Thread 0} \\
\text{Thread 1} \\
\text{Thread \ldots \text{THREADS-1}} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Columns 0:} \\
\text{Columns ((THREADS-1) \times M) / \text{THREADS}} \\
\text{Columns ((THREADS-1) \times M) / \text{THREADS}} \\
\end{array}
\]

\*Note: N and M are assumed to be multiples of THREADS

UPC Matrix Multiplication Code

```c
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4

shared [N*P / THREADS] int a[N][P];
shared [N*M / THREADS] int c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared [M/THREADS] int b[P][M];
void main (void) {
    int i, j, l; // private variables

    upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ; j++) {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
UPC Matrix Multiplication Code with Privatization

#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4

shared [N*P /THREADS] int a[N][P]; // N, P and M divisible by THREADS
shared [N*M /THREADS] int c[N][M];
shared[M/THREADS] int b[P][M];
int *a_priv, *c_priv;

void main (void) {
    int i, j, l; // private variables
    upc forall(i = 0 ; i<N ; i++; &c[i][0]) {
        a_priv = (int *)a[i]; c_priv = (int *)c[i];
        for (j=0 ; j<M ; j++) {
            c_priv[j] = 0;
            for (l= 0 ; l<P ; l++)
                c_priv[j] += a_priv[l]*b[l][j];
        }
    }
}

UPC Matrix Multiplication Code with block copy

#include <upc_relaxed.h>

shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared[M/THREADS] int b[P][M];
int b_local[P][M];

void main (void) {
    int i, j, l; // private variables
    for( i=0; i<P; i++ )
        for( j=0; j<THREADS; j++ )
            upc memget(&b_local[i][j*(M/THREADS)],
                        &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));
    upc forall(i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ; j++) {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++)
                c[i][j] += a[i][l]*b_local[l][j];
        }
    }
}
UPC Matrix Multiplication Code with Privatization and Block Copy

#include <upc_relaxed.h>
shared [N*P /THREADS] int a[N][P]; // N, P and M divisible by THREADS
shared [N*M /THREADS] int c[N][M];
shared[M/THREADS] int b[P][M];
int *a_priv, *c_priv, b_local[P][M];
void main (void) {
    int i, priv_i, j , l; // private variables
    for( i=0; i<P; i++ )
        for( j=0; j<THREADS; j++ )
            upc_memget(&b_local[i][j*(M/THREADS)],
                        &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));
    upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
        a_priv = (int *)a[i]; c_priv = (int *)c[i];
        for (j=0 ; j<M ;j++) {
            c_priv[j] = 0;
            for (l= 0 ; l<P ; l++)
                c_priv[j] += a_priv[l]*b_local[l][j];
        }
    }
}

Matrix Multiplication with dynamic memory

#include <upc_relaxed.h>
shared [N*P /THREADS] int *a;
shared [N*M /THREADS] int *c;
shared [M/THREADS] int *b;
void main (void) {
    int i, j , l; // private variables
    a=upc_all_alloc(THREADS,(N*P/THREADS)*upc_elemsizeof(*a));
    c=upc_all_alloc(THREADS,(N*M/THREADS)* upc_elemsizeof(*c));
    b=upc_all_alloc(P*THREADS, (M/THREADS)*upc_elemsizeof(*b));
    upc_forall(i = 0 ; i<N ; i++; &c[i*M]) {
        for (j=0 ; j<M ;j++) {
            c[i*M+j] = 0;
            for (l= 0 ; l<P ; l++)
                c[i*M+j] += a[i*P+l]*b[l*M+j];
        }
    }
}
Synchronization

- No implicit synchronization among the threads
- UPC provides the following synchronization mechanisms:
  - Barriers
  - Locks

Synchronization - Barriers

- No implicit synchronization among the threads
- UPC provides the following barrier synchronization constructs:
  - Barriers (Blocking)
    - upc_barrier expr opt;
  - Split-Phase Barriers (Non-blocking)
    - upc_notify expr opt;
    - upc_wait expr opt;
  Note: upc_notify is not blocking upc_wait is
Synchronization - Locks

• In UPC, shared data can be protected against multiple writers:
  - void upc_lock(upc_lock_t *l)
  - int upc_lock_attempt(upc_lock_t *l) //returns 1 on success and 0 on failure
  - void upc_unlock(upc_lock_t *l)

• Locks are allocated dynamically, and can be freed
• Locks are properly initialized after they are allocated

Dynamic lock allocation

• The locks can be managed using the following functions:
  • collective lock allocation (à la upc_all_alloc)
    upc_lock_t * upc_all_lock_alloc(void);
  • global lock allocation (à la upc_global_alloc)
    upc_lock_t * upc_global_lock_alloc(void)
  • lock freeing
    void upc_lock_free(upc_lock_t *ptr);
Collective lock allocation

- collective lock allocation

  upc_lock_t * upc_all_lock_alloc(void);

  - Needs to be called by all the threads
  - Returns a single lock to all calling threads

Global lock allocation

- global lock allocation

  upc_lock_t * upc_global_lock_alloc(void)

  - Returns one lock pointer to the calling thread
  - This is not a collective function
Lock freeing

- Lock freeing

```c
void upc_lock_free(upc_lock_t *l);
```

- This is not a collective function

Numerical Integration (computation of $\pi$)

- Integrate the function $f$ (which equals $\pi$)
Example: Using Locks in Numerical Integration

```c
#include <upc_relaxed.h>
#define N 1000000
#define f(x) 1/(1+x*x)

upc_lock_t *l;
shared float pi;

void main(void)
{
    float local_pi=0.0;
    int i;
    l = upc_all_lock_alloc();
    upc_barrier;
    upc_forall(i=0;i<N;i++; i)
        local_pi += (float) f((.5+i)/(N));
    local_pi *= (float) (4.0 / N);
    upc_lock(l); /*better with collectives*/
    pi += local_pi;
    upc_unlock(l);
    upc_barrier(); // Ensure all is done
    upc_lock_free( l );
    if(MYTHREAD==0)
        printf("PI=%f\n",pi);
}
```

Memory Consistency Models

- Has to do with ordering of shared operations, and when a change of a shared object by a thread becomes visible to others
- Consistency can be *strict* or *relaxed*
- Under the relaxed consistency model, the shared operations can be reordered by the compiler / runtime system
- The strict consistency model enforces sequential ordering of shared operations. (No operation on shared can begin before the previous ones are done, and changes become visible immediately)
Memory Consistency

- Default behavior can be controlled by the programmer and set at the program level:
  - To have strict memory consistency
    ```
    #include <upc_strict.h>
    ```
  - To have relaxed memory consistency
    ```
    #include <upc_relaxed.h>
    ```

Memory Consistency

- Default behavior can be altered for a variable definition in the declaration using:
  - Type qualifiers: `strict` & `relaxed`
- Default behavior can be altered for a statement or a block of statements using
  ```
  #pragma upc strict
  ```
  ```
  #pragma upc relaxed
  ```
- Highest precedence is at declarations, then pragmas, then program level
Memory Consistency- Fence

- UPC provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    • upc_fence;
  - UPC ensures that all shared references are issued before the upc_fence is completed

Memory Consistency Example

```c
strict shared int flag_ready = 0;
shared int result0, result1;

if (MYTHREAD==0)
{
    results0 = expression1;
    flag_ready=1; //if not strict, it could be
    // switched with the above statement
}

else if (MYTHREAD==1)
{
    while(!flag_ready); //Same note
    result1=expression2+results0;
}
```

- We could have used a barrier between the first and second statement in the if and the else code blocks. Expensive!! Affects all operations at all threads.
- We could have used a fence in the same places. Affects shared references at all threads!
- The above works as an example of point to point synchronization.