Productive Parallel Programming in PGAS: Unified Parallel C

• Montse Farreras – BSC Barcelona Supercomputing Center
  UPC Universitat Politècnica de Catalunya
• Calin Cascaval - IBM TJ Watson Research Center
• Gheorghe Almasi - IBM TJ Watson Research Center
• Ettore Tiotto - IBM Toronto Laboratory
• Kit Barton - IBM Toronto Laboratory
Outline

1. Overview of the PGAS programming model
2. Programming in Unified Parallel C
3. Scalability and performance considerations
4. Conclusions
1. Overview of the PGAS programming model

Some slides adapted with permission from Kathy Yelick
Motivation

• Current and next generation HPC architectures increase in size and complexity, which makes them challenging to program
  • Multicores, clusters of symmetric multiprocessors, heterogeneous accelerators (such as the Cell, GPUs), large core-count integrated machines (such as the Blue Gene),

• Parallelism is becoming ubiquitous, requiring high level programming models that satisfy a wide range of programmers and applications
  • Adequate programming tools offering ease of programming and productivity are essential
  • Users expect also good performance

• GOAL: Bridge the programmability GAP
  • Current Programming Models made for a few Expert Parallel Programmers
  • Our target is the large number of parallelism oblivious developers
Partitioned Global Address Space

- Partitioned Global Address Space (PGAS) languages provide a unique programming model to program large scale machines with easy-to-use shared memory paradigms.
- Explicitly parallel, shared-memory like programming model
- Control over the data layout

- **Global** addressable space
  - Allows programmers to declare and “directly” access data distributed across the machine
- **Partitioned** address space
  - Memory is logically partitioned between *local* and *remote* (a two-level hierarchy)
  - Forces the programmer to pay attention to data locality, by exposing the inherent NUMA-ness of current architectures

![Diagram of memory partitioning]
Computation is performed in multiple places.
- A place contains data that can be operated on remotely.
- Data lives in the place it was created, for its lifetime.

A datum in one place may point to a datum in another place.
- Data-structures (e.g. arrays) may be distributed across many places.

A place expresses locality.
PGAS languages

- Parallel language that follow the PGAS model:
  - **UPC** (C-based),
  - Co-Array Fortran (Fortran-based)
  - Titanium (Java-based, UC Berkeley)
  - X10 (IBM Java-based)
  - Chapel (Cray)
  - Fortress (Sun)
2.- Programming in Unified Parallel C
• Unified Parallel C (UPC) is:
  • An explicit parallel extension of ANSI C
  • A partitioned global address space language
  • Similar to the C language philosophy
  • Programmers are clever and careful, and may need to get close to hardware
    • to get performance, but
    • can get in trouble
  • Concise and efficient syntax
  • Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C
  • Based on ideas in Split-C, AC, and PCP
UPC Execution Model

• A number of threads working independently in a SPMD fashion
• Number of threads available as program variable THREADS
• MYTHREAD specifies thread index (0..THREADS-1)
• upc_barrier is a global synchronization: all wait
• There is a form of parallel loop that we will see later
• There are two compilation modes

• **Static Threads mode:**
  • THREADS is specified at compile time by the user (compiler option)
  • The program may use THREADS as a compile-time constant
  • The compiler generates more efficient code

• **Dynamic threads mode:**
  • Compiled code may be run with varying numbers of threads
  • THREADS is specified at runtime time by the user (via env. Variable UPC_NTHREADS)
Hello World in UPC

• Any legal C program is also a legal UPC program 😊
• If you compile and run it as UPC with N threads, it will run N copies of the program (Single Program executed by all threads).

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

int main() {
    printf("Thread \%d of \%d: Hello UPC world\n", MYTHREAD, THREADS);
    return 0;
}
```

```
hello > xlupc helloWorld.upc
hello > env UPC_NTHREADS=4 ./a.out
Thread 1 of 4: Hello UPC world
Thread 0 of 4: Hello UPC world
Thread 3 of 4: Hello UPC world
Thread 2 of 4: Hello UPC world
```
• Normal C variables and objects are allocated in the private memory space for each thread.

• Shared variables are allocated only once, with thread 0

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

• Shared variables may not be declared automatic, i.e., may not occur in a function definition, except as static. Why?
Data layout of Shared Arrays

• Shared arrays are spread over the threads, distributed in a cyclic fashion

  shared int x[THREADS]; /* 1 element per thread */
  shared int y[3][THREADS]; /* 3 elements per thread */
  shared int z[3][3]; /* 2 or 3 elements per thread */

• Assuming THREADS = 4:

Think of a linearized C array, then map round-robin on THREADS
int i, j, l;
shared int x[THREADS];
shared int y[3][THREADS];
shared int z[3][3];

Assuming THREADS = 4:

<table>
<thead>
<tr>
<th></th>
<th>Thread₀</th>
<th>Thread₁</th>
<th>Thread₂</th>
<th>Thread₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td>i, j, l</td>
<td>i, j, l</td>
<td>i, j, l</td>
<td>i, j, l</td>
</tr>
</tbody>
</table>
int i, j, l;
shared int x[THREADS];
shared int y[3][THREADS];
shared int z[3][3];

Assuming THREADS = 4:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀</td>
<td>y₀₀</td>
<td>z₀₀</td>
</tr>
<tr>
<td>x₁</td>
<td>y₀₁</td>
<td>z₁₁</td>
</tr>
<tr>
<td>x₂</td>
<td>y₀₂</td>
<td>z₀₁</td>
</tr>
<tr>
<td>x₃</td>
<td>y₀₃</td>
<td>z₁₀</td>
</tr>
</tbody>
</table>

shared

<table>
<thead>
<tr>
<th>i, j, l</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j, l</td>
</tr>
<tr>
<td>i, j, l</td>
</tr>
<tr>
<td>i, j, l</td>
</tr>
</tbody>
</table>
Example: Vector Addition

- **Questions about parallel vector additions:**
  - How to layout data (here it is cyclic)
  - Which processor does what (here it is “owner computes”)

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

shared int v1[N], v2[N], sum[N];
void main() {
  int i;
  for(i=0; i<N; i++)
    if (MYTHREAD == i%THREADS)
      sum[i]=v1[i]+v2[i];
}
```
The owner computes rule is very common in parallel programming
  • Loop over all; work on those owned by this proc
• UPC adds a special type of loop
  \[
  \text{upc\_forall}(\text{init}; \text{test}; \text{loop}; \text{affinity})
  \]
  \[
  \text{statement};
  \]
• Programmer indicates the iterations are independent
  • Undefined if there are dependencies across threads
• Affinity expression indicates which iterations to run on each thread. It may have one of two types:
  • **Integer**: \( \text{affinity}\%\text{THREADS} \text{ IS MYTHREAD} \)
    \[
    \text{upc\_forall}(i=0; i<N; ++i; i)
    \]
  • **Pointer**: \( \text{upc\_threadof(affinity)} \text{ IS MYTHREAD} \)
    \[
    \text{upc\_forall}(i=0; i<N; ++i; \&A[i])
    \]
Work Sharing with upc\_forall()

- Similar to C for loop, 4\textsuperscript{th} field indicates the affinity
- Thread that “owns” elem. A[i] executes iteration

```c
shared int sum[6], v1[6], v2[6];
upc\_forall(i=0; i < 6; i++; &sum[i]) {
    sum[i] = v1[i] + v2[i];
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>(v1)</th>
<th>(v2)</th>
<th>(\text{sum})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>0</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>1</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>2</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>3</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>4</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
<tr>
<td>5</td>
<td>(\text{th0})</td>
<td>(\text{th1})</td>
<td>(\text{No communication})</td>
</tr>
</tbody>
</table>
```c
#define N 100*THREADS

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

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

- Equivalent code could use “&sum[i]” for affinity
- Would the code be correct if the affinity expression were \texttt{i+1} rather than \texttt{i}?
UPC Global Synchronization

- Controls relative execution of threads
- UPC has two basic forms of barriers:
  - **Barrier**: block until all other threads arrive
    ```c
    UPC_barrier
    ```
  - **Split-phase barriers**
    ```c
    UPC_notify;  // this thread is ready for barrier
do computation unrelated to barrier
    UPC_wait;    // wait for others to be ready
    ```
- Optional labels allow for debugging
  ```c
  #define MERGE_BARRIER 12
  if (MYTHREAD%2 == 0) {
    ...
    UPC_barrier MERGE_BARRIER;
  } else {
    ...
    UPC_barrier MERGE_BARRIER;
  }
  ```
Summary: Shared Variables, Work sharing and Synchronization

• With what you’ve seen until now, you can write a bare-bones data-parallel program 😊

• Shared variables are distributed and visible to all threads
  • Shared scalars have affinity to thread 0
  • Shared arrays are distributed (cyclically by default) on all threads

• Execution model is SPMD with the upc_forall provided to share work
• Barriers and split barriers provide global synchronization

<table>
<thead>
<tr>
<th>Data</th>
<th>Work sharing</th>
<th>Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>upc_forall(i=0;i &lt; N; i ++; A[i])</td>
<td>upc_barrier</td>
</tr>
</tbody>
</table>
The cyclic layout is typically stored in one of two ways

- Distributed memory: each processor has a chunk of memory
  - Thread 0 would have: 0,THREADS, THREADS*2,… in a chunk
- Shared memory machine: all data may be on one chunk
  - Shared memory would have: 0,1,2,…THREADS,THREADS+1,…

Vector addition example can be rewritten as follows

```c
#define N 100*THREADS
shared [*] int v1[N], v2[N], sum[N];
void main() {
    int i;
    upc_forall(i=0; i<N; i++; &sum[i])
        sum[i]=v1[i]+v2[i];
}
```
Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>cyclic layout</td>
</tr>
<tr>
<td>[*]</td>
<td>blocked layout</td>
</tr>
<tr>
<td>[0] or []</td>
<td>indefinitely layout, all on 1 thread</td>
</tr>
<tr>
<td>[b] or [b1][b2]...[bn] = [b1<em>b2</em>...*bn]</td>
<td>fixed block size</td>
</tr>
</tbody>
</table>

- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element i has affinity with thread
  \[(i / \text{block}\_\text{size}) \% \text{THREADS}\]
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping
Distribution of a shared array in UPC

- Elements are distributed in block-cyclic fashion
- Each thread “owns” blocks of adjacent elements

```
shared [2] int X[10];

shared [*] int X[10];

shared [ ] int X[10];

shared [1] int X[10];
shared int X[10];
```
Distribution of a shared array in UPC

shared [2] int X[10];

shared [*] int X[10];

shared [ ] int X[10];

shared [1] int X[10];
shared int X[10];
Distribution of a shared array in UPC

2 threads

shared [2] int X[10];

Thread 0

Thread 1

shared [*] int X[10];

shared [] int X[10];

shared [1] int X[10];

shared int X[10];

Logical Distribution

Physical Distribution
Terminology

- `upc_threadof(&a[i])`
  - Thread that owns a[i]
- `upc_phaseof(&a[i])`
  - The position of a[i] within its block
- `course(&a[i])`
  - The block index of a[i]

**Examples**

- `upc_threadof(&a[2]) = 1`
- `upc_threadof(&a[5]) = 0`
- `upc_phaseof(&a[2]) = 0`
- `upc_phaseof(&a[5]) = 1`
- `course(&a[2]) = 0`
- `course(&a[5]) = 1`
Matrix-vector multiplication (matrix stored by rows)

- Which Blocking Factor (BF) is better?

```c
#define N THREADS
shared [BF] int a[N][N];
shared int b[N], c[N];

int main (void) {
    int i, j;
    upc_forall( i = 0; i < N; i++; i) {
        c[i] = 0;
        for (j = 0; j < N; j++)
            c[i] += a[i][j]*b[j];
    }
    return 0;
}
```
Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
  - Yes, UPC has pointers, use them if you dare …
- Here is the vector addition example using pointers

```c
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    shared int *p1, *p2;
    p1=v1; p2=v2;
    for (i=0; i<N; i++, p1++, p2++){
        if (i%THREADS == MYTHREAD)
            sum[i]= *p1 + *p2;
    }
}
```
Where does the pointer point to?

Where does the pointer reside?

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>PP ((p_1))</td>
<td>PS ((p_3))</td>
</tr>
<tr>
<td>Shared</td>
<td>SP ((p_2))</td>
<td>SS ((p_4))</td>
</tr>
</tbody>
</table>

- `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 to private is not recommended. Why?
UPC Pointers

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory: Fat pointers
Common Uses for UPC Pointer Types

```c
int *p1;
```

- These pointers are fast (just like C pointers)
- Use to access local data in part of code performing local work
- Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

```c
shared int *p2;
```

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication

```c
int * shared p3;
```

- Not recommended

```c
shared int * shared p4;
```

- Use to build shared linked structures, e.g., a linked list
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 shared to private is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast
Synchronization primitives

• We have seen upc_barrier, upc_notify and upc_wait
• UPC supports locks:
  • Represented by an opaque type: upc_lock_t
  • Must be allocated before use:
    upc_lock_t *upc_all_lock_alloc(void);
    allocates 1 lock, pointer to all threads, collective
    upc_lock_t *upc_global_lock_alloc(void);
    allocates 1 lock, pointer to all threads, called by one thread only
• To use a lock:
  void upc_lock(upc_lock_t *l)
  void upc_unlock(upc_lock_t *l)
  use at start and end of critical region
• Locks can be freed when not in use
  void upc_lock_free(upc_lock_t *ptr);
• UPC provides a fence construct
  • `upc_fence`
• UPC ensures that all accesses to share data issued before the `upc_fence` are complete
Dynamic memory allocation

- As in C, memory can be dynamically allocated
- UPC provides several memory allocation routines to obtain space in the shared heap
  - `shared void* upc_all_alloc(size_t nblocks, size_t nbytes)`
    - A collective operation that allocates memory on all threads
    - Layout equivalent to: `shared [nbytes] char[nblocks * nbytes]`
  - `shared void* upc_global_alloc(size_t nblocks, size_t nbytes)`
    - A non-collective operation, invoked by one thread to allocate memory on all threads
    - Layout equivalent to: `shared [nbytes] char[nblocks * nbytes]`
  - `shared void* upc_alloc(size_t nbytes)`
    - A non-collective operation to obtain memory in the thread’s shared section of memory
  - `void upc_free(shared void *p)`
    - A non-collective operation to free data allocated in shared memory
Dynamic memory allocation

- **upc_all_alloc**
  - a collective
  - shared [nbytes] char[nblocks * nbytes]

- **upc_global_alloc**
  - A non-collective
  - shared [nbytes] char[nblocks * nbytes]

- **upc_alloc**
  - A non-collective
  - in the thread’s shared section of memory
Distributed arrays allocated dynamically

typedef shared [] int *sdblptr;
shared sdblptr directory[THREADS];

int main() {
  ...
  directory[MYTHREAD] = upc_alloc(local_size*sizeof(int));
  upc_barrier;
  ...
  /* use the array */
  upc_barrier;
  upc_free(directory[MYTHREAD]);
}

Diagram:

```
  P_0  P_0  P_2  P_2

  TH 0  TH 1  TH 2  TH n

  directory
```

Legend:

- **SHARED**
- **PRIVATE**
Data movement

- Fine grain (array element, by array element access) are easy to program in an imperative way. However, especially on distributed memory machines, block transfers are more efficient.

- UPC provides library functions for data movement and collective operations:
  - **upc_memset**
    - Set a block of values in shared memory
  - **upc_memget, upc_memput**
    - Transfer blocks of data from shared memory to/from private memory
  - **upc_memcpy**
    - Transfer blocks of data from shared memory to shared memory
  - Collective operations (broadcast, reduce, etc.)
    - A set of function calls is specified in the standard, but it’s being reworked. In next specification UPC 1.3 this calls are asynchronous
      - **upc_memcpy_async, upc_wait**
Memory Consistency

• The consistency model defines the order in which one thread may see another threads accesses to memory
  
  • If you write a program with unsynchronized accesses, what happens?
  
  • Does this work?

    ```
    data = ...             while (!flag) { };
    flag = 1;             ... = data;   // use the data
    ```

• UPC has two types of accesses:

  • **Strict**: will always appear in order
  
  • **Relaxed**: may appear out of order to other threads

• There are several ways of designating the type, commonly:

  • Use the include file:

    ```
    #include <upc_relaxed.h>
    ```

    • Which makes all accesses in the file relaxed by default

  • Use `strict` on variables that are used as synchronization (flag)

    ```
    strict shared int flag;
    ```
Data movement Collectives

- Used to move shared data across threads:
  - `upc_all_broadcast(shared void* dst, shared void* src, size_t nbytes, ...)`
    - A thread copies a block of memory it “owns” and sends it to all threads
  - `upc_all_scatter(shared void* dst, shared void *src, size_t nbytes, ...)`
    - A single thread splits memory in blocks and sends each block to a different thread
  - `upc_all_gather(shared void* dst, shared void *src, size_t nbytes, ...)`
    - Each thread copies a block of memory it “owns” and sends it to a single thread
  - `upc_all_gather_all(shared void* dst, shared void *src, size_t nbytes, ...)`
    - Each threads copies a block of memory it “owns” and sends it to all threads
  - `upc_all_exchange(shared void* dst, shared void *src, size_t nbytes, ...)`
    - Each threads splits memory in blocks and sends each block to all thread
  - `upc_all_permute(shared void* dst, shared void *src, shared int* perm, size_t nbytes, ...)`
    - Each threads copies a block of memory and sends it to thread in perm[i]
Thread 0 copies a block of memory and sends it to all threads
Thread 0 sends a unique block to all threads
Each thread sends a block to thread 0
copies a block of memory from one shared memory area with affinity to the \( i \)th thread to the \( i \)th block of a shared memory area on each thread.
The upc all exchange function copies the $i$th block of memory from a shared memory area that has affinity to thread $j$ to the $j$th block of a shared memory area that has affinity to thread $i$. 
Computational Collectives

• Used to perform data reductions
  - upc_all_reduceT(shared void* dst, shared void* src, upc_op_t op, …)
  - upc_all_prefix_reduceT(shared void* dst, shared void *src, upc_op_t op, …)

• One version for each type T (22 versions in total)

• Many operations are supported:
  • OP can be: +, *, &, |, xor, &&, ||, min, max
  • perform OP on all elements of src array and place result in dst array
Threads perform partial sums, each partial sum added and result stored on thread 0
UPC is the best !!
Considerations about Collectives in UPC Standard specification

- Collective operations are scalable 😊
- They define synchronization modes 😊
- They operate on shared data 😊

- They are “single-valued”: all UPC threads are required to submit the same buffer as an argument (only a single shared array can be involved in a collective at a time) X
- No notion of communicators X
- The synchronization modes are confusing X
- The users miss MPI_Allreduce, MPI_AllToAllv
  - Limited usability X
  - Bad productivity X

UPC Collectives library 2.0 coming soon! (Planned for UPC 1.4)
Summary

• UPC designed to be consistent with C
  • Some low level details, such as memory layout are exposed
  • Ability to use pointers and arrays interchangeably
• Designed for high performance
  • Memory consistency explicit
  • Small implementation

■ UPC specification (current 1.2) and other documents
  
  http://upc.gwu.edu

  UPC 1.3 specification is currently being reworked, more info in:
  
  http://code.google.com/p/upc-specification/issues/list
QUICK Feedback

- I would like to get some feedback from you concerning productivity, performance, scalability, flexibility, usability …
- Think about your favorite program, think about how to program it in UPC
- You should be able to have an idea in 5 min.
- Write in a piece of paper
  - 3 strengths
  - 3 weaknesses

Of UPC language
3. Scalability and performance considerations
Scalability: Rules of thumb

• Things to avoid:
  • UPC tempts user into fine-grain communication
  • UPC tempts user into bad data layouts
  • The siren song of UPC locks

• Things to take advantage of:
  • Global view makes reasoning about program easier
    • The “G” in PGAS
    • Shared memory model -> productivity
    • Affinity concept -> performance
  • Collective communication
Examples

• Example 1: Vector addition
• Example 2: Simple sum
• Example 3: Stencil
• Example 4: Matrix Multiplication
• Example 5: Molecular dynamics
EXAMPLE 1: Vector addition

\[ r_i = \sum v1_i \times v2_i \]

shared int sum[6], v1[6], v2[6];
upc_forall(i=0; i < 6; i++; &sum[i]) {
    sum[i] = v1[i] + v2[i];
}

<table>
<thead>
<tr>
<th>i</th>
<th>v1</th>
<th>v2</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 threads

Embarrassingly parallel

Good scalability

No communication
EXAMPLE 2: Simple sum: Introduction

- Simple forall loop to add values into a sum

```c
shared int values[N], sum;

sum = 0;
upc_forall (int i=0; i<N; i++; &values[i])
    sum += values[i];
```

- Is there a problem with this code?
- Implementation is broken: “sum” is not guarded by a lock
- Write-after-write hazard; will get different answers every time
Simple sum: using locks

- Easy you say ... let’s use a lock!

```c
shared int values[N], sum;
upc_lock_t mylock;

upc forall (int i=0; i<N; i++; &values[i]) {
    upc_lock (&mylock);
    sum += values[i];
    upc_unlock (&mylock);
}
```

- Correct implementation 😊😊😊

- But horrible performance 😞😞😞

- Lock is used for every array value !!!
Better performance if $N \gg \text{THREADS}$

```
shared int values[N], sum;
shared int partialsums[THREADS];

partialsum[MYTHREAD] = 0;
upc forall (int i=0; i<N; i++; &values[i])
    partialsum[MYTHREAD] += values[i];

upc forall (int i=0; i<THREADS; i++; partialsums[i])
    upc_lock (&mylock);
    sum += partialsums[i];
    upc_unlock (&mylock);
```
- Access to shared data (even if local) is typically more expensive than access to private data.
  - Some compilers perform optimizations to privatize local shared data

```c
shared int values[N], sum;
shared int partialsums[THREADS];
int localsum;

localsum=0;
upc_forall (int i=0; i<N; i++; &values[i]) {
    localsum += values[i];
}
partialsum[MYTHREAD] = localsum;
upc_forall (int i=0; i<THREADS; i++; partialsums[i]) {
    upc_lock (&mylock);
    sum += partialsums[i];
    sum += partialsums[i];
    upc_unlock (&mylock);
}
```
Simple sum: avoid locks

- Better performance if $N \gg THREADS$
- Still $O(THREADS)$ communication!

```c
shared int values[N], sum;
shared int partialsums[THREADS];
int localsum=0;

upc_forall (int i=0; i<N; i++; &values[i]) {
    localsum += values[i];
}
partialsun[MYTHREAD] = localsum;
upc_barrier;
if(MYTHREAD==0)
    for(int i=0; i<THREADS; i++){
        sum +=partialsun[i];
    }
```
Simple sum: avoid locks and use collective

- $O(\log(\text{THREADS}))$ communication!

```c
shared int values[N], sum;
shared int partialsums[THREADS];
int localsum=0;

upc_forall (int i=0; i<N; i++; &values[i]) {
    localsum += values[i];
}
partialsum[MYTHREAD]=localsum;

upc_all_reduceI(&sum,
    partialsum,
    UPC_ADD,
    THREADS,
    1,
    NULL,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```
Simple sum: just use a collective

- Assuming $N = k \times \text{THREADS}$ (or array padded with zeroes)

  ```c
  shared int values[N], sum;
  upc_all_reduceI (&sum,
                  values,
                  UPC_ADD,
                  N,
                  N/THREADS,
                  NULL,
                  UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
  ```

- Typical $O(\log(\text{THREADS}))$ scalability (like MPI reductions)

- What is the best option?

- Your lesson: avoid locks! There is almost always a better solution
EXAMPLE 3: Access Granularity: Stencil

Naive solution:

```c
shared double A[N][N];

upc_forall (i=1; i<N-1; i++; continue)
  upc_forall (j=1; j<N-1; j++; &A[i][j])
```

Communication traffic:
4 * N * N elements
4 * N * N accesses

This is bad because all read memory accesses are likely non-local
Access Granularity: Banded Stencil

Better solution: banded layout

shared \([N*N/THREADS]\) double A[N][N];

upc_forall (i=1; i<N-1; i++; continue)
  upc_forall (j=1; j<N-1; j++; &A[i][j])

Better, because only 2*N accesses per thread are non-local
Access Granularity: Shadow Exchange

Banded layout with shadow regions:

```c
#define B (N+2*THREADS)/THREADS
shared [B*N] double A[N+2*THREADS][N];

/* exchange shadows (code incomplete, no bounds checks!) */
int l=MYTHREAD*B;     /* lower shadow */
upc_memget (&A[l][0], &A[l-2][0], N*sizeof(double));

int u=(MYTHREAD+1)*B-1; /* upper shadow row */
upc_memget (&A[u][0], &A[u+2][0], N*sizeof(double));

/* stencil code as usual */
...
```

Communication traffic:
2 * N * THREADS elements
2 * THREADS accesses
**EXAMPLE 3: Access Granularity: Tiled layout**

**Tiled layout (UPC extension)**

```c
#define B
shared [B][B] double A[N][N];
```

- Very complicated code (exchange buffers are not contiguous) (*)
- Highly scalable: per-thread communication costs decrease with scaling

(*) compiler aggregation optimization can help keep code small

**Communication traffic:**
- $4 \times N \times \sqrt{T}$ elements
- $4 \times T$ accesses
EXAMPLE 4: Matrix multiplication: Introduction

shared double A[M][P], B[P][N], C[M][N];

forall (i=0; i<M; i++; continue)
    forall (j=0; j<N; j++; &C[i][j])
        for (k=0; k<P; k++)
            C[i][j] += A[i][k]*B[k][j];

Problem:
• Accesses to A and B are mostly non-local
• Fine grain remote access == bad performance!

\[ C_{ij} = \sum A_{ik} \times B_{kj} \]
Matrix multiplication: Block matrices

shared [B][B] A[M][P], B[P][N], C[M][N];

forall (i=0; i<M; i++; continue)
  forall (j=0; j<N; j++; &C[i][j])
    for (k=0; k<P; k++) {
      upc_memget (alocal, &A[i][k], B*B*sizeof(double));
      upc_memget (blocal, &B[k][j], B*B*sizeof(double));
      dgemm (alocal, blocal, &C[i][j]);
    }

• Good:
  • Fewer accesses, large granularity
  • Improved single-node performance (ESSL library call)

• Bad:
  • Code has changed significantly
  • Still not scalable performance: O(n^3) communication
Blocked Matrix Multiply scaling
P5 cluster, 4 nodes x 8 threads/node

Number of threads vs. Performance (GFlops)

- 4 nodes
- 4 nodes-ideal
Matrix multiplication: New Layout

typedef shared { int x[B][B]; } Block;
shared [P1] Block A[M1][P1];
shared [N1] Block B[P1][N1];
shared [N1] Block C[M1][N1];

Good:

• no locality issues on A and C (traversing the same way)

Bad:

• B is traversed across block layout (communication!)
Matrix Multiplication: Tiled Layouts

```c
#pragma processors C(Tx, Ty)
shared [B][B] double C[M][N];
```

- **Good:**
  - Allows control of block placement on processor grid
  - Allows C to be accessed as array, not as struct
  - Allows communication among rows, cols of processors (scalable communication)

- **Bad:**
  - UPC extension: not available in vanilla UPC
  - Not yet available in IBM UPC

- **Good:**
  - Attempting to add this into standard

**WARNING!!!**
Research Work Not part of the UPC standard
Scalability: Matrix multiplication: Tiled layout key to performance

UPC matrix multiplication on a 16-rack Blue Gene/L
• N particles in 3D space interact with each other
• Compute particles new position, velocity, acceleration at each time step

- **GOOD:**
  Given force acting on each particle the computation of particles position, velocity, acceleration is an embarrassing parallel problem

- **BAD:**
  Force acting on particle p.f[i] is a function of the gravitational attraction of all other particles …
Molecular Dynamics

N time steps

Compute forces acting on particles

Compute particles position, velocity, acceleration

Result

typedef struct {
    double p[3];  // particle position
    double v[3];  // particle velocity
    double a[3];  // particle acceleration
    double f[3];  // force acting on particle
} particle_t;

#define BF (NPARTS/THREADS)

shared [BF] particle_t PARTS[NPARTS];
shared [BF] double POT[NPARTS];
shared [BF] double KIN[NPARTS];

upc_forall(int i = 0; i < NPARTS; i++; &PARTS[i])
    for(int j = 0; j < NDIM; j++) {
        PARTS[i].p[j] += PARTS[i].v[j]*dt + 0.5*dt*dt*PARTS[i].a[j];
        PARTS[i].v[j] += 0.5*dt*(PARTS[i].f[j]*rmass + PARTS[i].a[j]);
        PARTS[i].a[j] = PARTS[i].f[j]*rmass;
    }
Molecular Dynamics

Initial State

Compute forces acting on particles

N time steps

Compute particles position, velocity, acceleration

Result

upc_forall (i = 0; i < NPARTS; i++; &PARTS[i]) {
  ...
  for (j = 0; j < NPARTS; j++) {
    if (i != j) {
      d = dist(&PARTS[i].p[0], &PARTS[j].p[0], rij);
      POT[i] += 0.5*V(d);
      for (k = 0; k < NDIM; k++)
        PARTS[i].f[k] = PARTS[i].f[k] - rij[k]*DV(d)/d;
    }
  }
}

Remote accesses

double dist (shared double *r1, shared double *r2, double *dr) {
  for (int i=0; i < NDIM; i++) {
    dr[i] = r1[i] - r2[i];
    d += dr[i]*dr[i];
  }
  return sqrt(d);
}
Molecular Dynamics

Initial State

Compute forces acting on particles

N time steps

Compute particles position, velocity, acceleration

Result

upc_forall (i = 0; i < NPARTS; i++; &PARTS[i]) {
    particle_t *p = (particle_t*) &PARTS[i];
    ...
    for (j = 0; j < NPARTS; j++) {
        if (i != j) {
            d = dist_local(&p->pos[0], &PARTS[j].pos[0], rij);
            POT[i] = POT[i] + 0.5*V(d);
            for (k = 0; k < NDIM; k++)
                p->f[k] = p->f[k] - rij[k]*DV(d)/d;
        }
    }
}

double dist_local(double *r1, shared double *r2, double *dr) {
    for (int i=0; i < NDIM; i++) {
        dr[i] = r1[i] - r2[i];
        d += dr[i]*dr[i];
    }
    return sqrt(d);
}

Further Improvements:
- Prefetch r2[i] with upc_memget?
- Prefetch PARTS[j] in caller?
4. Conclusions
Currently there are both free and commercial UPC compilers available, for a broad spectrum of architectures. In detail

- HP UPC (commercial).
- Cray UPC (commercial). Not very mature. It uses the GASNet communication system.
- GCC UPC (free): developed by Intrepid, is supports a lot of architectures (from x86 to Cray) and it uses the Berkeley runtime
- Berkeley UPC (free): fully portable source-to-source compiler implementation. It uses the GASNet communication system.
- XLUPC IBM (commercial) It supports Blue Gene and PowerPC SMP's running AIX or Linux. It provides interesting language extensions 😊

It is active, specification 1.3 to appear next month. The committee is already working on 1.4 and 2.0
Conclusions

**UPC = Performance + Productivity**

**Performance**
- Exploitation of data locality
- Coalescing of communication
- Overlapping communication and computation
- One-sided communication
- Optimized collective library

**Productivity**
- Simple syntax based on C
- Easy partitioning of shared data
- Work-sharing construct with locality information
- No explicit need to manage communication with function calls
- Simple thread synchronization

UPC standard is being reworked! Your ideas can be taken
Thanks!

Questions?