CS 267
Unified Parallel C (UPC)

Kathy Yelick

http://upc.lbl.gov

Slides adapted from some by Tarek El-Ghazawi (GWU)
UPC Outline

1. Background
2. UPC Execution Model
3. Basic Memory Model: Shared vs. Private Scalars
4. Synchronization
5. Collectives
6. Data and Pointers
7. Dynamic Memory Management
8. Programming Examples
8. Performance Tuning and Early Results
9. Concluding Remarks
Context

• Most parallel programs are written using either:
  • Message passing with a SPMD model
    • Usually for scientific applications with C++/Fortran
    • Scales easily
  • Shared memory with threads in OpenMP, Threads+C/C++/F or Java
    • Usually for non-scientific applications
    • Easier to program, but less scalable performance
• Global Address Space (GAS) Languages take the best of both
  • global address space like threads (programmability)
  • SPMD parallelism like MPI (performance)
  • local/global distinction, i.e., layout matters (performance)
Partitioned Global Address Space Languages

- Explicitly-parallel programming model with SPMD parallelism
  - Fixed at program start-up, typically 1 thread per processor
- Global address space model of memory
  - Allows programmer to directly represent distributed data structures
- Address space is logically partitioned
  - Local vs. remote memory (two-level hierarchy)
- Programmer control over performance critical decisions
  - Data layout and communication
- Performance transparency and tunability are goals
  - Initial implementation can use fine-grained shared memory
- Multiple PGAS languages: UPC (C), CAF (Fortran), Titanium (Java)
Global Address Space Eases Programming

- The languages share the global address space abstraction
  - Shared memory is logically partitioned by processors
  - Remote memory may stay remote: no automatic caching implied
  - One-sided communication: reads/writes of shared variables
  - Both individual and bulk memory copies
- Languages differ on details
  - Some models have a separate private memory area
  - Distributed array generality and how they are constructed
Current Implementations of PGAS Languages

- A successful language/library must run everywhere
- **UPC**
  - Commercial compilers available on Cray, SGI, HP machines
  - Open source compiler from LBNL/UCB (source-to-source)
  - Open source gcc-based compiler from Intrepid
- **CAF**
  - Commercial compiler available on Cray machines
  - Open source compiler available from Rice
- **Titanium**
  - Open source compiler from UCB runs on most machines
- **Common tools**
  - Open64 open source research compiler infrastructure
  - ARMCI, GASNet for distributed memory implementations
  - Pthreads, System V shared memory
UPC Overview and Design Philosophy

- Unified Parallel C (UPC) is:
  - An explicit parallel extension of ANSI C
  - A partitioned global address space language
  - Sometimes called a GAS 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
UPC Execution Model

• A number of threads working independently in a SPMD fashion
  • Number of threads specified at compile-time or run-time; 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
    • The program may use THREADS as a compile-time constant
  • Dynamic threads mode:
    • Compiled code may be run with varying numbers of threads
Hello World in UPC

• Any legal C program is also a legal UPC program
• If you compile and run it as UPC with P threads, it will run P copies of the program.
• Using this fact, plus the identifiers from the previous slides, we can parallel hello world:

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

main() {
    printf("Thread %d of %d: hello UPC world\n", MYTHREAD, THREADS);
}
```
Example: Monte Carlo Pi Calculation

• Estimate Pi by throwing darts at a unit square
• Calculate percentage that fall in the unit circle
  • Area of square = \( r^2 = 1 \)
  • Area of circle quadrant = \( \frac{1}{4} \pi r^2 = \pi/4 \)
• Randomly throw darts at \( x,y \) positions
• If \( x^2 + y^2 < 1 \), then point is inside circle
• Compute ratio:
  • \# points inside / \# points total
  • \( \pi = 4 \times \text{ratio} \)
Pi in UPC

- Independent estimates of pi:

```c
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;

    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);

    srand(MYTHREAD*17);

    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables
Each thread can use input arguments
Initialize random in math library
Each thread calls “hit” separately
## Helper Code for Pi in UPC

**• Required includes:**
```c
#include <stdio.h>
#include <math.h>
#include <upc.h>
```

**• Function to throw dart and calculate where it hits:**
```c
int hit(){
    int const rand_max = 0xFFFFFFFF;
    double x = ((double) rand()) / RAND_MAX;
    double y = ((double) rand()) / RAND_MAX;
    if ((x*x + y*y) <= 1.0) {
        return(1);
    } else {
        return(0);
    }
}
```
Shared vs. Private Variables
Private vs. Shared Variables in UPC

• 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; // use sparingly: performance
  int mine;
  ```

• Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static. Why?

![Diagram showing private and shared variables in UPC threads]

3/2/2007 CS267 Lecture: UPC
Pi in UPC: Shared Memory Style

• Parallel computing of pi, but with a bug

```c
shared int hits;
main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        hits += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
        printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

shared variable to record hits
divide work up evenly
accumulate hits

What is the problem with this program?
Shared Arrays Are Cyclic By Default

- Shared scalars always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads

```c
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 */
```

- In the pictures below, assume THREADS = 4
- Red elts have affinity to thread 0

Think of linearized C array, then map in round-robin

As a 2D array, y is logically blocked by columns

z is not
Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
    ... declarations an initialization code omitted
    for (i=0; i < my_trials; i++)
        all_hits[MYTHREAD] += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
        for (i=0; i < THREADS; i++)
            hits += all_hits[i];
        printf("PI estimated to %f.\n", 4.0*hits/trials);
    }
}
```
UPC Synchronization
UPC Global Synchronization

- 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;
  }
  ```
Synchronization - Locks

- Locks in UPC are represented by an opaque type:
  upc_lock_t
- Locks must be allocated before use:
  upc_lock_t *upc_all_lock_alloc(void);
    allocates 1 lock, pointer to all threads
  upc_lock_t *upc_global_lock_alloc(void);
    allocates 1 lock, pointer to one thread
- 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);
Parallel computing of pi, without the bug

shared int hits;

main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        my_hits += hit();
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*hits/trials);
}
Recap: Private vs. Shared Variables in UPC

We saw several kinds of variables in the pi example:

- Private scalars (my_hits)
- Shared scalars (hits)
- Shared arrays (all_hits)
- Shared locks (hit_lock)

<table>
<thead>
<tr>
<th>Global address space</th>
<th>Thread₀</th>
<th>Thread₁</th>
<th>...</th>
<th>Threadₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>my_hits</td>
<td>my_hits</td>
<td>...</td>
<td>my_hits</td>
</tr>
<tr>
<td></td>
<td>hit_lock</td>
<td>hit_lock</td>
<td></td>
<td>hit_lock</td>
</tr>
<tr>
<td>shared</td>
<td>all_hits[0]</td>
<td>all_hits[1]</td>
<td>...</td>
<td>all_hits[n]</td>
</tr>
</tbody>
</table>

where:

n = Threads - 1

Shared

Private
UPC Collectives
UPC Collectives in General

• The UPC collectives interface is in the language spec:
  - http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf
• It contains typical functions:
  - Data movement: broadcast, scatter, gather, …
  - Computational: reduce, prefix, …
• Interface has synchronization modes:
  - Avoid over-synchronizing (barrier before/after is simplest semantics, but may be unnecessary)
  - Data being collected may be read/written by any thread simultaneously
• Simple interface for collecting scalar values (int, double, …)
  - Berkeley UPC value-based collectives
  - Works with any compiler
  - http://upc.lbl.gov/docs/user/README-collectivev.txt
Pi in UPC: Data Parallel Style

- The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
- Use a reduction for better scalability

```c
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
...
for (i=0; i < my_trials; i++)
    my_hits += hit();
my_hits = // type, input, thread, op
    bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
// upc_barrier;
if (MYTHREAD == 0)
    printf("PI: %f", 4.0*my_hits/trials);
}
```

Berkeley collectives
- no shared variables
- barrier implied by collective
UPC (Value-Based) Collectives in General

- General arguments:
  - `rootthread` is the thread ID for the root (e.g., the source of a broadcast)
  - All 'value' arguments indicate an l-value (i.e., a variable or array element, not a literal or an arbitrary expression)
  - All 'TYPE' arguments should the scalar type of collective operation
  - `upc_op_t` is one of: UPC_ADD, UPC_MULT, UPC_AND, UPC_OR, UPC_XOR, UPC_LOGAND, UPC_LOGOR, UPC_MIN, UPC_MAX

- Computational Collectives
  - `TYPE bupc_allv_reduce(TYPE, TYPE value, int rootthread, upc_op_t reductionop)`
  - `TYPE bupc_allv_reduce_all(TYPE, TYPE value, upc_op_t reductionop)`
  - `TYPE bupc_allv_prefix_reduce(TYPE, TYPE value, upc_op_t reductionop)`

- Data movement collectives
  - `TYPE bupc_allv_broadcast(TYPE, TYPE value, int rootthread)`
  - `TYPE bupc_allv_scatter(TYPE, int rootthread, TYPE *rootsrcarray)`
  - `TYPE *bupc_allv_gather(TYPE, TYPE value, int rootthread, TYPE *rootdestarray)`
    - Gather a 'value' (which has type TYPE) from each thread to 'rootthread', and place them (in order by source thread) into the local array 'rootdestarray' on 'rootthread'.
  - `TYPE *bupc_allv_gather_all(TYPE, TYPE value, TYPE *destarray)`
  - `TYPE bupc_allv_permute(TYPE, TYPE value, int tothreadid)`
    - Perform a permutation of 'value's across all threads. Each thread passes a value and a unique thread identifier to receive it - each thread returns the value it receives.
Full UPC Collectives: Synchronization

• When can a collective argument begin executing?
  • Arguments with affinity to thread $i$ are ready when thread $i$ calls the function; results with affinity to thread $i$ are ready when thread $i$ returns.
  • This is appealing but it is incorrect: In a broadcast, thread 1 does not know when thread 0 is ready.
**UPC Collective: Sync Flags**

- In full UPC Collectives, blocks of data may be collected
- A extra argument of each collective function is the sync mode of type upc_flag_t.
- Values of sync mode are formed by or-ing together a constant of the form UPC_IN_XSYNC and a constant of the form UPC_OUT_YSYNC, where X and Y may be NO, MY, or ALL.
- If sync_mode is (UPC IN_XSYNC | UPC OUT_YSYNC), then if X is:
  - NO the collective function may begin to read or write data when the first thread has entered the collective function call,
  - MY the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
  - ALL the collective function may begin to read or write data only after all threads have entered the collective function call
- and if Y is
  - NO the collective function may read and write data until the last thread has returned from the collective function call,
  - MY the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete3, and
  - ALL the collective function call may return only after all reads and writes of data are complete.
Work Distribution Using upc_forall
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];
}
```

• cyclic layout
• owner computes
Work Sharing with upc_forall()

- The idiom in the previous slide is very common
  - Loop over all; work on those owned by this proc
- UPC adds a special type of loop
  
  ```
  upc_forall(init; test; loop; affinity)
  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: `affinity%THREADS is MYTHREAD`
  - Pointer: `upc_threadof(affinity) is MYTHREAD`
- Syntactic sugar for loop on previous slide
  - Some compilers may do better than this, e.g.,
    ```
    for(i=MYTHREAD; i<N; i+=THREADS)
    ```
  - Rather than having all threads iterate N times:
    ```
    for(i=0; i<N; i++) if (MYTHREAD == i%THREADS)
    ```
Vector Addition with upc_forall

• The **vadd** example can be rewritten as follows
  • Equivalent code could use “&sum[i]” for affinity
  • The code would be correct but slow if the affinity expression were **i+1** rather than **i**.

```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];
}
```

The cyclic data distribution may perform poorly on some machines
Distributed Arrays in UPC
Blocked Layouts in UPC

- If this code were doing nearest neighbor averaging (3pt stencil) the cyclic layout would be the worst possible layout.
- Instead, want a blocked layout
- Vector addition example can be rewritten as follows using a blocked layout

```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:
  • Empty (cyclic layout)
  • [*] (blocked layout)
  • [0] or [] (indefinite layout, all on 1 thread)
  • [b] or [b1][b2]…[bn] = [b1*b2*…bn] (fixed block size)
• 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
  \[(i \div \text{block\_size}) \mod \text{THREADS}\]
• In 2D and higher, linearize the elements as in a C representation, and then use above mapping
2D Array Layouts in UPC

- Array a1 has a row layout and array a2 has a block row layout.
  
  ```
  shared [m] int a1 [n][m];
  shared [k*m] int a2 [n][m];
  ```

- If \((k + m) \% \text{THREADS} = = 0\) then a3 has a row layout
  
  ```
  shared int a3 [n][m+k];
  ```

- To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.

- Assume \(r*c = \text{THREADS}\);
  
  ```
  shared [b1][b2] int a5 [m][n][r][c][b1][b2];
  ```

- or equivalently
  
  ```
  shared [b1*b2] int a5 [m][n][r][c][b1][b2];
  ```
UPC Matrix Vector Multiplication Code

• Matrix-vector multiplication with matrix stored by rows
• (Contrived example: problems size is P x P)

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

void main (void) {
    int i, j, l;
    upc_forall( i = 0 ; i < THREADS ; i++; i) {
        c[i] = 0;
        for ( l= 0 ; l < THREADS ; l++)
            c[i] += a[i][l]*b[l];
    }
}
/* mat_mult_1.c */
#include <upc_relaxed.h>

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

shared [N*P /THREADS] int a[N][P], c[N][M];
// a and c are row-wise blocked shared matrices

shared[M/THREADS] int b[P][M]; //column-wise blocking

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];
        }
    }
}
Notes on the Matrix Multiplication Example

- The UPC code for the matrix multiplication is almost the same size as the sequential code.
- Shared variable declarations include the keyword `shared`.
- Making a private copy of matrix B in each thread might result in better performance since many remote memory operations can be avoided.
- Can be done with the help of `upc_memget`.
Domain Decomposition for UPC

- Exploits 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:

\[\begin{align*}
\text{Thread 0} & : 0 \ldots ((N*P) / \text{THREADS}) - 1 \\
\text{Thread 1} & : (N*P) / \text{THREADS} \ldots (2*N*P) / \text{THREADS} - 1 \\
\text{Thread } \text{THREADS}-1 & : ((\text{THREADS}-1) \times N*P) / \text{THREADS} \ldots (\text{THREADS}*N*P) / \text{THREADS} - 1
\end{align*}\]

\[\begin{align*}
\text{Columns 0: } & (M/\text{THREADS}) - 1 \\
\text{Columns } (\text{THREAD}-1) \times M/\text{THREADS};(M-1) & : (\text{THREAD}-1) \times M/\text{THREADS};(M-1)
\end{align*}\]

- \textbf{Note: } \( N \) and \( M \) are assumed to be multiples of \( \text{THREADS} \)
Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
- Here is the vector addition example using pointers

```
#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;
}
```
### UPC Pointers

#### Where does the pointer point?

<table>
<thead>
<tr>
<th>Where does the pointer reside?</th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>PP (p1)</td>
<td>PS (p3)</td>
</tr>
<tr>
<td>Shared</td>
<td>SP (p2)</td>
<td>SS (p4)</td>
</tr>
</tbody>
</table>

#### Code Examples

```c
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.
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 */

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.
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
shared int *p2;
• Use to refer to remote data
• Larger and slower due to test-for-local + possible communication
int *shared p3;
• Not recommended
shared int *shared p4;
• Use to build shared linked structures, e.g., a linked list
```
UPC Pointers

• 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>Virtual Address</th>
<th>Thread</th>
<th>Phase</th>
</tr>
</thead>
</table>

• Example: Cray T3E implementation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Thread</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>49</td>
<td>48</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>0</td>
</tr>
</tbody>
</table>
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 pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast
Special Functions

• size_t upc_threadof(shared void *ptr); returns the thread number that has affinity to the pointer to shared
• size_t upc_phaseof(shared void *ptr); returns the index (position within the block) field of the pointer to shared
• shared void *upc_resetphase(shared void *ptr); resets the phase to zero
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
Global Memory Allocation

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

- \textbf{nblocks} : number of blocks
- \textbf{nbytes} : block size

- Non-collective: called by one thread
- The calling thread allocates a contiguous memory space in the shared space
- If called by more than one thread, multiple regions are allocated and each thread which makes the call gets a different pointer
- Space allocated per calling thread is equivalent to :
  \texttt{shared [nbytes] char[nblocks * nbytes]}
Collective Global Memory Allocation

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

- nblocks: 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]`
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
Distributed Arrays Directory Style

• Some high performance UPC programmers avoid the UPC style arrays
  • Instead, build directories of distributed objects
  • Also more general

typedef shared [] double *sdblptr;
shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));
upc_barrier;
Memory Consistency in UPC

• The consistency model defines the order in which one thread may see another thread's accesses to memory
  • If you write a program with unsynchronized accesses, what happens?
  • Does this work?
    ```c
    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:
    ```c
    #include <upc_relaxed.h>
    ```
  • Which makes all accesses in the file relaxed by default
  • Use strict on variables that are used as synchronization (flag)
Synchronization - Fence

- Upc provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    - upc_fence;
  - UPC ensures that all shared references issued before the upc_fence are complete
Performance of UPC
PGAS Languages have Performance Advantages

Strategy for acceptance of a new language
• Make it run faster than anything else

Keys to high performance
• Parallelism:
  • Scaling the number of processors
• Maximize single node performance
  • Generate friendly code or use tuned libraries (BLAS, FFTW, etc.)
• Avoid (unnecessary) communication cost
  • Latency, bandwidth, overhead
  • Berkeley UPC and Titanium use GASNet communication layer
• Avoid unnecessary delays due to dependencies
  • Load balance; Pipeline algorithmic dependencies
One-Sided vs Two-Sided

one-sided put message

| address       | data payload |

| message id    | data payload |

two-sided message

- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth
One-Sided vs. Two-Sided: Practice

• InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
• Half power point (N ½ ) differs by one order of magnitude
• This is not a criticism of the implementation!

NERSC Jacquard machine with Opteron processors

Joint work with Paul Hargrove and Dan Bonachea
GASNet: Portability and High-Performance

GASNet better for latency across machines

3/2/2007

CS267 Lecture: UPC

Joint work with UPC Group; GASNet design by Dan Bonachea
GASNet: Portability and High-Performance

GASNet at least as high (comparable) for large messages
GASNet: Portability and High-Performance

GASNet excels at mid-range sizes: important for overlap

Joint work with UPC Group; GASNet design by Dan Bonachea
Case Study: NAS FT

• Performance of Exchange (Alltoall) is critical
  • 1D FFTs in each dimension, 3 phases
  • Transpose after first 2 for locality
  • Bisection bandwidth-limited
    • Problem as #procs grows

• Three approaches:
  • Exchange:
    • wait for 2\textsuperscript{nd} dim FFTs to finish, send 1 message per processor pair
  • Slab:
    • wait for chunk of rows destined for 1 proc, send when ready
  • Pencil:
    • send each row as it completes
Overlapping Communication

• Goal: make use of “all the wires all the time”
  • Schedule communication to avoid network backup
• Trade-off: overhead vs. overlap
  • Exchange has fewest messages, less message overhead
  • Slabs and pencils have more overlap; pencils the most
• Example: Class D problem on 256 Processors

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Exchange (all data at once)</td>
<td>512 Kbytes</td>
</tr>
<tr>
<td>Slabs (contiguous rows that go to 1 processor)</td>
<td>64 Kbytes</td>
</tr>
<tr>
<td>Pencils (single row)</td>
<td>16 Kbytes</td>
</tr>
</tbody>
</table>
NAS FT Variants Performance Summary

Best MFlop rates for all NAS Fortran/MPI
Best MPI (always Slabs)
Best UPC (always Pencils)

Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea

.5 Tflops
Case Study: LU Factorization

- Direct methods have complicated dependencies
  - Especially with pivoting (unpredictable communication)
  - Especially for sparse matrices (dependence graph with holes)
- LU Factorization in UPC
  - Use overlap ideas and multithreading to mask latency
  - Multithreaded: UPC threads + user threads + threaded BLAS
    - Panel factorization: Including pivoting
    - Update to a block of U
    - Trailing submatrix updates
- Status:
  - Dense LU done: HPL-compliant
  - Sparse version underway
UPC HPL Performance

Comparison to ScaLAPACK on an Altix, a 2 x 4 process grid
- ScaLAPACK (block size 64) 25.25 GFlop/s (tried several block sizes)
- UPC LU (block size 256) - 33.60 GFlop/s, (block size 64) - 26.47 GFlop/s
- \( n = 32000 \) on a 4x4 process grid
  - ScaLAPACK - 43.34 GFlop/s (block size = 64)
  - UPC - 70.26 Gflop/s (block size = 200)

Joint work with Parry Husbands
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
• Berkeley compiler (used for next homework)
  http://upc.lbl.gov
• Language specification and other documents
  http://upc.gwu.edu