Unified Parallel C

Yili Zheng
Research Scientist
Computational Research Department
Lawrence Berkeley National Lab
Outline

• Overview of UPC
• How does a UPC implementation work
• Examples
• Optimization tips and good practices
• Summary of tools and references
Partitioned Global Address Space In UPC

- Global data view abstraction for productivity
- Vertical partitions among threads for locality control
- Horizontal partitions between shared and private segments for data placement optimizations
- Friendly to non-cache-coherent architectures
One-Sided vs. Two-Sided Messaging

Two-sided messaging
- Message does not contain information about the final destination; need to look it up on the target node
- Point-to-point synchronization implied with all transfers

One-sided messaging
- Message contains information about the final destination
- Decouple synchronization from data movement

Two-sided message (e.g., MPI)
- message id
- data payload

One-sided put (e.g., UPC)
- dest. addr.
- data payload
Overview of Unified Parallel C

- C99 extension (PGAS C)
  - Partitioned Global Address Space for data sharing
  - One-sided communication (Put/Get, Read/Write)
  - Loop-level parallelism (upc_forall)

- SPMD execution model
  - Total number of threads in the execution: THREADS
  - My thread id (0,…,THREADS-1): MYTHREAD

- Widely available
  - Open source: Berkeley UPC, GCC UPC
  - Commercial: Cray, IBM, HP, SGI
  - Platforms: Shared-memory, Ethernet, Infiniband, Cray, IBM, …
Why Use UPC?

- Pros
  - A global address space for shared-memory programming
  - One-sided communication is a good match for hardware RDMA
  - Can safely reuse non-pthread-safe legacy sequential libraries

- Cons
  - Memory consistency model is complicated
    - Good news: most users don’t need to worry for common use patterns
  - Performance tuning is as hard as other programming models
Example: Hello World

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

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

```
> upcc helloworld.upc
> upcrun -n 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
```
How to use UPC on Cray XE / XK

- module swap PrgEnv-pgi PrgEnv-cray
- cc -h upc helloworld.upc
- aprun -n 8 ./a.out
UPC is simple

• 5 necessary keywords:
  • shared
  • upc_fence // non-collective
  • upc_barrier // imply a fence
  • THREADS
  • MYTHREAD

• Communication is implicit
  • shared int s;
  • s = 5; // write (put)
  • a = s; // read (get)
Sharing Data

- Static shared data defined in file scope
  - shared int j; /* shared scalar variable resides on thread 0 */
  - shared int a[10]; /* shared array distributed in round-robin */
- Shared arrays are distributed in a 1-D block-cyclic fashion over all threads
  - shared [blocking_factor] int array[size];
  - Example: shared [2] int b[12]; on 4 UPC threads
    - logical data layout
      - physical data layout
Data Layouts in a Nutshell

• Static non-array objects have affinity with thread 0
• Array layouts are controlled by the blocking factor:
  • Empty or [1] (cyclic layout)
    \[ \text{shared int} == \text{shared [1] int} \]
  • [*] (blocked layout)
    \[ \text{shared [*] int a[sz]} == \text{shared [sz/THREADS] int a[sz]} \]
  • [0] or [] (indefinite layout, all on 1 thread)
    \[ \text{shared [] int} == \text{shared [0] int} \]
  • [b] (fixed block size, aka block-cyclic)
• The affinity of an array element \( A[i] \) is determined by:
  \[ (i / \text{block\_size}) \% \text{THREADS} \]
• M-D arrays linearize elements in row-major format
UPC Pointers

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 */
Multi-Dimensional Arrays

Static 2-D array:  shared [*] double A[M][N];

Dynamic 2-D array:  shared [] double **A;

A and pointers can be private and replicated on all threads.
Loop level parallelism

- `upc forall(init; test; loop; affinity) statement;`
- `upc forall` is a collective operation in which, for each execution of the loop body, the controlling expression and affinity expression are single-valued.
- Programmer asserts that the iterations are independent
- Affinity expression indicates which iterations will run on each thread. It may have one of two types:
  - Integer: `(affinity%THREADS) == MYTHREAD`
  - Pointer: `upc_threadof(affinity) == MYTHREAD`
    ```
    upc forall(i=0; i<N; i++; i) stmt;
    ```
    equivalent to
    ```
    for(i=0; i<N; i++)
        if (MYTHREAD == i % THREADS) stmt;
    ```
Synchronization - Locks

- Locks in UPC are represented by an opaque type:
  ```c
  upc_lock_t
  ```
- Locks must be allocated before use:
  ```c
  upc_lock_t *upc_all_lock_alloc(void);
  ```
  collective call - allocates 1 lock, same pointer to all threads
  ```c
  upc_lock_t *upc_global_lock_alloc(void);
  ```
  non-collective - allocates 1 lock per caller
- To use a lock:
  ```c
  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
  ```c
  void upc_lock_free(upc_lock_t *ptr);
  ```
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;
  }
  ```
Bulk Data Movement and Nonblocking Communication

- Loops to perform element-wise data movement could potentially be slow because of network traffic per element
- Language introduces variants of memcpy to address this issue:
  - `upc_memcpy (shared void * restrict dst,
               shared const void * restrict src, size_t n)`
  - `upc_memput (shared void * restrict dst,
                const void * restrict src, size_t n)`
  - `upc_memget (void * restrict dst,
                shared const void * restrict src, size_t n)`
Data Movement Collectives

- `upc_all_broadcast`(shared void* `dst`, shared void* `src`, size_t `nbytes`, ...)
- `upc_all_scatter`(shared void* `dst`, shared void *`src`, size_t `nbytes`, ...)
- `upc_all_gather`(shared void* `dst`, shared void *`src`, size_t `nbytes`, ...)
- `upc_all_gather_all`(shared void* `dst`, shared void *`src`, size_t `nbytes`, ...)
- `upc_all_exchange`(shared void* `dst`, shared void *`src`, size_t `nbytes`, ...)
- `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]`
Computational Collectives

upc_all_reduceT(shared void* dst, shared void* src, upc_op_t op, …)

data type T: char, short, int, float, double, long long double,…

upc_op_t: +, *, &, |, xor, &&, ||, min, max

upc_all_reduceT computes:

\[
\sum_{i=0}^{7} A[i] = 0 1 2 3 4 5 6 7
\]

upc_all_prefix_reduceT(shared void* dst, shared void *src, upc_op_t op, …)
Example: Jacobi (5-point stencil)

shared [ngrid*ngrid/THREADS] double u[ngrid][ngrid];
shared [ngrid*ngrid/THREADS] double unew[ngrid][ngrid];
shared [ngrid*ngrid/THREADS] double f[ngrid][ngrid];

upc_forall( int i=1; i<n; i++ ) {
    for(int j=1; j<n; j++) {
        utmp = 0.25 * (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - h*h*f[i][j]); /* 5-point stencil */
        unew[i][j] = omega * utmp + (1.0-omega)*u[i][j];
    }
}

- Good spatial locality
- Mostly local memory accesses
- No explicit communication ghost-zone management
Example: Random Access (GUPS)

```c
shared uint64 Table[TableSize]; /* cyclic distribution */
uint64 i, ran;

/* owner computes, iteration matches data distribution */
upc_forall (i = 0; i < TableSize; i++; i) Table[i] = i;

upc_barrier; /* synchronization */

ran = starts(NUPDATE / THREADS * MYTHREAD); /* ran. seed */

for (i = MYTHREAD; i < NUPDATE; i+=THREHDAS) /* SPMD */
{
    ran = (ran << 1) ^ (((int64_1) ran < 0) ? POLY : 0);
    Table[ran & (TableSize-1)] = Table[ran & (TableSize-1)] ^ ran;
}
upc_barrier; /* synchronization */
```

The MPI version is about 150 lines due to message aggregation.
UPC Compiler Implementation

Source-to-source translator

- Pros: portable
- Cons: may lose program information in two-phase compilation
- Example: Berkeley UPC

Source-to-object-code compiler

- Pros: easier to implement
- UPC specific optimization
- Cons: less portable
- Example: GCC UPC and most vendor UPC

UPC code

UPC source-to-source translator

C code

UPC code

UPC source-to-object code compiler

Assembly code
Programming models on BlueGene/P

CAF Apps

CAF Compiler

CAF Runtime

GASNet DCMF Conduit

UPC Apps

BUPC Compiler

BUPC Runtime

IBM XL UPC Compiler

IBM PGAS Runtime

IBM DCMF Messaging Library

BlueGene/P Networks (Torus, Collective and Barrier)

MPI Apps

MPICH2 BGP Port

Unified Parallel C -- Yili Zheng
Berkeley UPC Software Stack

UPC Applications

UPC-to-C Translator

Translated C code with Runtime Calls

UPC Runtime

GASNet Communication Library

Network Drivers and OS Libraries
Translation and Call Graph Example

```c
shared [] int * shared sp;
*sp = a;
```

- **UPC-to-C Translator**
  - `UPCR_PUT_PSHARED_VAL(sp, a);`

- **UPC Runtime**
  - Is `*sp` local?
    - No: `gasnet_put(sp, a);`
    - Yes: `memcpy(sp, a);`

- **GASNet**
- **Memory Access**
Casting Shared-Pointer to Local

Kernel code of the STREAM benchmark using shared-pointers

shared [] double *sa, *sb, *sc;
for (i=0; i<nelems; i++) {
    sa[i] = sb[i] + alpha * sc[i];
}

Kernel code of the STREAM benchmark using local pointers

shared [] double *sa, *sb, *sc;
double *a, *b, *c;
a=(double *)sa; b=(double *)sb; c=(double *)sc;
for (i=0; i<nelems; i++) {
    a[i] = b[i] + alpha * c[i];
}
Shared Data Access Performance: Local Pointer vs. Pointer-to-shared

**Shared Data Access Time on 32-core AMD**

- Local pointer
- Pointer-to-shared Berkeley UPC
- Pointer-to-shared GCCUPC

**Shared Data Access Time on 8-core Intel**

- Local pointer
- Pointer-to-shared Berkeley UPC
- Pointer-to-shared GCCUPC
Use Physical Shared-Memory for Inter-Process Communication

- Cast a pointer-to-shared affined to another thread but can be accessed directly by hardware load and store
  - void * upc_cast(shared void *ptr);
  - Castability query:
    - int upc_castable(shared void *ptr);
    - int upc_thread_castable(unsigned int threadnum);

- Implemented by cross-mapping physical memory to virtual address spaces of all processes sharing the node

- Save memory space and copy overheads that would be otherwise introduced by bounce-buffers
Memory Consistency Models

- UPC supports two memory consistency models: strict and relaxed
- **Strict consistency**
  - **Usage:** `#pragma upc strict` or `strict shared [ ] double *sa;`
  - Provide a total ordering for all memory accesses
  - Easy to reasoning about but takes a huge performance penalty
- **Relaxed consistency**
  - **Usage:** `#pragma upc relaxed` or `relaxed shared [ ] double *sa;`
  - Allow concurrent and out-of-order data accesses within a synchronization phase
  - Deliver better performance but may introduce data races if synchronization is done correctly
- **In practice**
  - Use the relaxed consistency model (default) until encountering errors
  - Use the strict consistency model for testing and debugging
Memory Consistency Performance: Relaxed vs. Strict

**Shared Data Access Time on 32-Core AMD**

- **Relaxed consistency**
- **Strict consistency**

**Shared Data Access Time on 8-Core Intel**

- **Relaxed consistency**
- **Strict consistency**
Example: Matrix Transpose

shared double *sa, *sb;
size_t N;

correct but very poor:

Global array view may tempt you to use a naïve implementation

Correct but very poor performance:
- All fine-grained accesses
- No data locality
- Difficult to vectorize

```c
upc_forall(i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        ij = i*N+j;
        ji = j*N+i;
        sb[ij] = sa[ji];
    }
}
```
Example: Optimized Matrix Transpose

- Use a block data layout
- Transpose data blocks by a collective operation
- Transpose the elements in the block locally

```c
B = N/THREADS;
nbytes = sizeof(double)*B*B;
upc_all_exchange(sb, sa, nbytes, UPC_IN_MYSYNC|UPC_OUT_MYSYNC);

/* local transpose */
for (t=0; t<THREADS; t++) {
    la = (double *)&sa[MYTHREAD] + B*B*t;
    lb = (double *)&sb[MYTHREAD] + B*B*t;
    local_transpose(la, lb, B);
}
```
Matrix Transpose Performance

Transpose on 32-Core AMD

Transpose on 8-Core Intel
Example: Matrix Multiplication

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

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

- Naïve implementation is very slow
  - Many fine-grained remote accesses
  - Recurring overheads in access through pointers-to-shared
  - Do not have optimization for the sequential part, such as register blocking, cache blocking and vectorization

- But it is really simply to write if you don’t care about performance (such as in prototyping or non-critical path)
Optimized UPC Parallel DGEMM

- 2-D block-cyclic data layout
- Use parallel algorithms such as SUMMA
- Transfer data in large blocks
- Use optimized BLAS dgemm (e.g., Intel MKL)
- Use non-blocking collective communication if available (e.g., row and column broadcasts)
Matrix Multiplication Performance

32-Core AMD (Opteron 8387)

8-Core Intel (Xeon E5530)

UPC with ACML
ACML with OpenMP

UPC with Intel MKL
Intel MKL with OpenMP
Example: 3-D FFT

- 2-D Data Partitioning
- Row-column algorithm with overlapping local FFT and transpose (all-to-all communication)
- UPC non-blocking operations enabled fine-grained overlapping for better performance
# FFT Performance on Multi-Core

Performance of 3D-FFT (512x256x256) on 32-core AMD (Mflops)

<table>
<thead>
<tr>
<th>Threads</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFTW</td>
<td>4561.3</td>
<td>7338.7</td>
<td>8756.4</td>
<td>8365.5</td>
</tr>
<tr>
<td>UPC with</td>
<td>2306.61</td>
<td>4242.28</td>
<td>7210.87</td>
<td>9849.7</td>
</tr>
<tr>
<td>FFTW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuesday, July 24, 12
Unified Parallel C -- Yili Zheng 39
FFT Performance on BlueGene/P

3-D FFT Weak Scaling Performance on IBM BlueGene/P

- Ideal Peak
- UPC Collectives (Slabs)
- UPC Collectives (Packed Slabs)
- MPI Collectives (Packed Slabs)

GFlops vs. Cores

Good
Pitfalls in Programming with UPC

- Abuse fine-grained inter-node data accesses – generate tons of tiny data packets
- Flood data from many to one – congest the network
- Share everything and access data uniformly – forget about data localities and NUMA issues
- Use excessive locking/unlocking – lock operations are expensive, especially on distributed-memory systems
- Hand code common math functions (instead of using optimized libraries such as BLAS, FFTW, INTEL MKL, IBM ESSL, …)

Performance Penalty!
UPC Programming Tips

• Use local pointer to access the local part of shared data by casting pointer-to-shared to local pointer
• Leverage data affinity information and manage shared data layout to minimize remote accesses (both inter-node and NUMA)
• Use non-blocking communication if available
• Use collectives
• Use remote atomic operations if available
UPC one for two?

- Hybrid Programming Styles with UPC
  - fine-grained (shared memory style)
  - bulk synchronous (message passing style)

- Hybrid Execution with UPC
  - Map UPC threads hierarchically to groups of Pthreads
    - Threads within a process share resources and the same virtual address space
    - Processes within a node use physically shared memory for fast communication
    - Inter-node communication uses the network
  - Balance resource sharing and isolation
    - Too much sharing: resource contention (lower performance), prone to race conditions
    - No sharing: resource idling (lower throughput)
Interoperability: Mix it up

- UPC with other sequential languages: C++, FORTRAN
- MPI with UPC
  - Each MPI process is also a UPC thread
  - Each MPI process spawns a few UPC threads. MPI for inter-process communication and UPC for intra-process communication
- UPC with OpenMP
  - Map each UPC thread to an OS process and spawn OpenMP threads
- UPC with CUDA and OpenCL
  - Similar to MPI + CUDA/OpenCL
UPC 1.3

• Coming this Fall
• Main features
  • Non-blocking memory copy operations
    • Implicit non-blocking memory operations – fire and forget
    • upc_memcpy_nbi(...);
    • upc_fence;
  • UPC atomics
    • CAS
    • Op
    • Fetch and Op
  • High precision timers
  • Collective memory deallocation (upc_all_free)
• Many bug fixes and clarifications
• http://code.google.com/p/upc-specification/
Tools

- **Eclipse Parallel Tools Platform (PTP)**
- **Parallel Performance Wizard (PPW)**
  - [http://ppw.hcs.ufl.edu/](http://ppw.hcs.ufl.edu/)
- **GDB UPC**
- **Totalview**
- **Distributed Debugging Tool (DDT) from Allinea Software**
- **All other parallel computing tools for multi-process and multi-thread programs**
  - Executing a UPC program is just like running a normal multi-process/multi-thread program from the OS’s perspective.
Resources and Contacts

• Web sites:
  • UPC community portal: http://upc.gwu.edu
  • IBM XL UPC: http://www.alphaworks.ibm.com/tech/upccompiler
  • GCC UPC: http://www.gccupc.org
  • Berkeley UPC: http://upc.lbl.gov

• Email lists:
  • UPC Mailing Lists: http://upc.gwu.edu/upc_mail_group.html
  • public Berkeley UPC users list: upc-users@lbl.gov
  • Berkeley UPC/GASNet developers: upc-devel@lbl.gov
THANK YOU!