ScaleUPC: A UPC Compiler for Multi-Core Systems

Weiming Zhao
Michigan Technological University
Motivations

- Multi-core computing becomes ubiquitous
  - efficient parallel computing

- Current multi-threaded programming often uses a conventional imperative language plus the pthread library
  - high complexity
  - low productivity
  - poor portability

- PGAS languages have already shown advantages in large-scale parallel computing
  - programmer friendly

Efficient resource utilization for UPC on multi-core computers
ScaleUPC

- Proof-of-concept UPC-to-C compiler/runtime
  - Based on an open source compiler: Scale (A Scalable Compiler for Analytical Experiments)
  - Converts each UPC thread to a pthread
  - Supports clusters (using MPI) / Hybrid mode
UPC Specific Constructs

- UPC thread
  - pthread
- upc_forall
  - for(;;) if (MYTHREAD==threadof()) {...}
- upc_lock_t
  - pthread_mutex_t
- upc_barrier
  - pthread_cond_wait / pthread_cond_broadcast
- upc_all_alloc
  - upc_barrier() ; if(MYTHREAD==THREAD_0) malloc();
Data Storage - Scalars, Private Arrays

- Shared/Private, Global/Local Data Objects
  - (global) shared -> C global variables
    - allocated in heap, naturally shared by all threads
  - Global private (global scope but not shared)
    - each thread keeps a separate instance
      1. allocated in Thread Local Storage
      2. declared as globals but modified by "__thread"
      3. declared as global array and referenced by MYTHREAD
  - Local (private) -> C local variables
Pointer Representation

- pointers to private data: C's built-in pointers
- pointers to shared data:
  - **UPC-specific operations**: `upc_threadof()`, `upc_phaseof()`, `upc_blocksizeof()`, `upc_elemsizeof()`
  - **Pointers-to-shared operations**: `p++`, `p2-p1`, `(shared [2] T*)p`
  - **struct**
    1. block address: the base address of a block
    2. thread: the thread affinity of the pointed-to object
    3. phase: the offset within the block
    4. block number: the logical block number (facilitates the calculation of pointer distance);
    5. type: type info (the block size specified in declaration or dynamic allocation, the size of each element, etc.)
Data Storage - Shared Arrays

shared [2] int a[10]; (THREADS=3)

Logical Layout

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[7]</td>
<td>a[9]</td>
<td></td>
</tr>
</tbody>
</table>

addr:

0 1 2 3 4 5 6 7 8 9
val: [0] [1] [6] [7] [2] [3] [8] [9] [4] [5]

thread major

addr:

0 1 2 3 4 5 6 7 8 9
val: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

block major
Comparison of the Two Layouts

**Thread-Major Layout**
- Follows UPC semantics when the pointers are privatized
- Inefficient:
  - 15 operations for general pointers ($p+i$)
  - 8 operations for array references ($a+i$)
  - needs more temp variables

![Thread-Major Layout Diagram]

**Block-Major Layout**
- Quick pointer arithmetic
  - same cost as regular C pointers
- Privatized pointers breaks UPC semantics when crossing block boundary
  - requires extra run-time support
Processor Scheduling

- Multi-core Processor shares some resources
  - L2/L3 Cache
  - Memory Bus

- Memory Bus Sharing
  - compete for bandwidth -> delay

- Cache Sharing
  - Conflicts
  - Improves hit ratio

Ideally, CPU affinity should be dynamically selected based on memory access patterns e.g. 2 xeon quad-cores
Profiling-Based Scheduling

- Static, profiling-based algorithm
  1. Given a training input, run the program with different binding policies respectively
  2. For each run, collect the cache hit rate
     - HW counter / software instrumentation
  3. Compare the average memory access cost
     Choose "shared cache" binding if
     1. hit ratio is improved and
     2. the effect of improved cache hits outweighs the slow down due to bus contention
        - access time $T = (1 - \text{miss rate}) \times \text{hit time} + \text{miss rate} \times \text{penalty}$
        - $T_1 < T_2 \Rightarrow \frac{\text{miss rate}_1}{\text{miss rate}_2} < \frac{(\text{Penalty}_2 - \text{hit time})}{(\text{Penalty}_1 - \text{hit time})}$
        - Penalty = Bus Time + Memory Access Time
        - When bus is shared, assume 2x Bus Time
Experimental Results

- OS: Linux 2.6
- CPU: 2x Xeon 2.33 GHz Quad-Core processors (8 cores)
- Compilers:
  - Intrepid UPC 4.0.3.4 (-O3)
  - Berkeley UPC 2.6.0 (-O)
  - GNU OpenMP compiler 4.1.2
- Benchmarks
  - Matrix Multiplication (2048 * 2048 integers)
  - N-Queen (GWU) (16 * 16)
  - NAS NPB (class B)  
    - UPC version (2.0)
    - CG, EP, IS, FT, MG
    - O0, O1 (privatization), O3 (O1 + remote prefetching)
    - OpenMP version
Execution Time

MM, NQ and NPB O0 on Xeon / 8 threads

NPB O3 on Xeon / 8 threads

- block-major / thread-major: 3.16
- block-major / Berkeley: 2.24
- block-major / Intrepid: 2.25
- thread-major / Intrepid: 1.07
- thread-major / Berkeley: 0.7
Execution Time

ScaleUPC / Berkeley: 1.23
ScaleUPC / Intrepid: 0.94 (CG)
ScaleUPC / OpenMP: 0.98

NPB O3 on Xeon / 8 threads
Scalability
Effects of Profiling-Based Static Scheduling

- training run
  - data size:
    - class A for NPB
    - 1024 × 1024 arrays for MM
    - 8 × 8 chess board for NQ
  - 2 threads, block major
  - cache simulation for "shared" / "dedicated" L2
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L2 Misses (%)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>50.0</td>
<td>99.9</td>
</tr>
<tr>
<td>NQ</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CG O0</td>
<td>25.9</td>
<td>25.9</td>
</tr>
<tr>
<td>CG O3</td>
<td>25.9</td>
<td>25.9</td>
</tr>
<tr>
<td>EP O0</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>FT O0</td>
<td>12.8</td>
<td>12.8</td>
</tr>
<tr>
<td>FT O1</td>
<td>12.9</td>
<td>12.8</td>
</tr>
<tr>
<td>IS O0</td>
<td>28.4</td>
<td>42.4</td>
</tr>
<tr>
<td>IS O1</td>
<td>36.5</td>
<td>36.3</td>
</tr>
<tr>
<td>MG O0</td>
<td>62.9</td>
<td>43.3</td>
</tr>
<tr>
<td>MG O3</td>
<td>63.6</td>
<td>44.9</td>
</tr>
</tbody>
</table>

**Mean**

<table>
<thead>
<tr>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sep. Pac.</td>
</tr>
<tr>
<td>1.03</td>
</tr>
</tbody>
</table>
4-thread results

● 3 binding schemes:
  ○ dedicated L2
  ○ shared L2 in separated package
  ○ shared L2 in the same package (share bus)

● Our profiling-based model still works
  ○ MM - 38% speed-up with shared L2
  ○ IS O0 - 11% speed-up with shared L2
  ○ Others: 0-12% slow down with shared L2
void norm2u3(...)  
{ ...  
upc_lock(critical_lock);  

  s += p_s;  
if (p_a > max) 
  max = p_a;  

upc_unlock(critical_lock);  

upc_barrier 20;  

*rnm2 = sqrt(s/(double)n);  
*rnmu = max;  

if (MYTHREAD == 0)  
{  
  max = 0;  
  s = 0.0;  
} ...  
}
Related Work

- Berkely UPC
  - supports pthreads as an optimization for SMP systems
  - suffers from the overhead of address arithmetic

- Intrepid UPC
  - UPC thread as a process
Conclusion

- For UPC on multi-core, the cost of pointer-to-shared arithmetic becomes non-trivial
- Memory performance is also important
  - Scheduling impacts performance