Efficient Data Race Detection for Unified Parallel C

ParLab Winter Retreat
1/14/2011

Costin Iancu, LBL
Nick Jalbert, UC Berkeley
Chang-Seo Park, UC Berkeley
Koushik Sen, UC Berkeley
Motivation

- Unified Parallel C (UPC)
  - Uniform parallel programming model for both shared and distributed memory hardware
  - Single Program Multiple Data (SPMD) + Partitioned Global Address Space (PGAS)

- Limited tool support for correctness of UPC programs
  - Unique challenges for UPC not in previous work on testing concurrent programs

- Build a tool for testing and debugging parallel UPC programs with the following goals
  - Scalable (works for large real-world programs)
  - Identify small candidate set of bugs (prune false positives)
  - Reproducible
Our Previous Work: Active Testing

- Leverage program analysis to make testing quickly find real concurrency bugs
  - Phase 1: Use imprecise static or dynamic program analysis to find *bug patterns* where a potential concurrency bug can happen (Race Detector)
  - Phase 2: *Directed testing* to confirm potential bugs as real (Race Tester)

- CalFuzzer: Active testing framework for Java
  - Largest program analyzed: Jigsaw (500KLOC)

- Thrille: Active testing framework for C/C++ with pthreads
  - Largest program analyzed: x264 (38KLOC)
Active Testing Cartoon: Phase I

Potential Collision
Active Testing Cartoon: Phase II
Differences and Challenges for UPC

- Java and pthreads programs
  - Synchronization with locks and condition variables
  - Single node

- UPC has different programming model (SPMD)
  - Large scale
  - Bulk communication
  - Collective operations with data movement
  - Memory consistency
Example Data Race in UPC

- Simple matrix vector multiply

\[
\begin{pmatrix}
\text{c} \\
\text{A} \\
\text{b}
\end{pmatrix} = \begin{pmatrix}
\text{A} \\
\times
\end{pmatrix}
\]
Example Data Race in UPC

- Simple matrix vector multiply

```c
1: void matvec(shared [N] int A[N][N], shared int B[N],
               shared int C[N]) {
2:     upc_forall(int i = 0; i < N; i++; &C[i]) {
3:         int sum = 0;
4:         for(int j = 0; j < N; j++)
5:             sum += A[i][j] * B[j];
6:         C[i] = sum;
7:     }
8:}
```

- Works as intended for matvec(A, B, C) where B ≠ C
Example Data Race in UPC

- Simple matrix vector multiply

```c
1: void matvec(shared [N] int A[N][N], shared int B[N],
    shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:        int sum = 0;
4:        for(int j = 0; j < N; j++)
5:            sum += A[i][j] * B[j];
6:        C[i] = sum;
7:    }
8:}
```

- Works as intended for `matvec(A, B, C)` where `B ≠ C`
- What about `matvec(A, B, B)`?
Example Data Race in UPC

- Simple matrix vector multiply

1: void matvec(shared [N] int A[N][N], shared int B[N],
               shared int C[N]) {
2:    upc_forall(int i = 0; i < N; i++; &C[i]) {
3:       int sum = 0;
4:       for(int j = 0; j < N; j++)
5:           sum += A[i][j] * B[j];
6:       C[i] = sum;
7:    }
8:}

- Works as intended for matvec(A, B, C) where B ≠ C
- What about matvec(A, B, B)?
- Results may vary for each execution
Thread Interposition Library and Lightweight Extensions
  - Framework for active testing UPC programs

Instrument UPC source code at compile time
  - Using macro expansions, add hooks for analyses

Phase 1: Race detector
  - Observe execution and predict which accesses may potentially have a data race

Phase 2: Race tester
  - Re-execute program while controlling the scheduler to create actual data race scenarios predicted in phase 1
Phase 1: Race Detection

- Data race between statements $s_1$ and $s_2$ when
  - Threads $t_1$ and $t_2$ are about to execute $s_1$ and $s_2$, respectively, which access the same memory location and one of the accesses is a write

- Runtime tracking of shared memory accesses
  - Store all shared memory accesses in a database
  - Keep track of locks held and barrier phase

- Search for conflicting shared memory accesses
  - Find pairs of memory accesses from different threads not protected with a common lock (Eraser)
  - The access pair “may happen in parallel”

- Output a list of statement pairs that may have a data race
Phase 1: Tuple Generation

- Generate tuple \((m, t, L, a, p, s)\) for each shared memory access where
  - \(m\): shared memory address range
  - \(t\): thread accessing \(m\)
  - \(L\): set of locks held by \(t\)
  - \(a\): type of access (Read or Write)
  - \(p\): barrier phase of \(t\)
  - \(s\): statement label (line number) for access
Phase 1: Tuple Generation

1: void matvec(shared [N] int A[N][N], shared int B[N], shared int C[N]) {
2:     upc_forall(int i = 0; i < N; i++; &C[i]) {
3:         int sum = 0;
4:         for(int j = 0; j < N; j++)
5:             sum += A[i][j] * B[j];
6:         C[i] = sum;
7:     }
8:}

Example assumes B and C are aliased
Phase 1: Checking for Conflicts

- Two accesses $e_1 = (m_1, t_1, L_1, a_1, p_1, s_1)$ and $e_2 = (m_2, t_2, L_2, a_2, p_2, s_2)$ are in conflict when
  - memory range overlaps \( (m_1 \cap m_2 \neq \emptyset) \)
  - accesses from different threads \( (t_1 \neq t_2) \)
  - no common locks held \( (L_1 \cap L_2 = \emptyset) \)
  - at least one write \( (a_1 = \text{Write} \lor a_2 = \text{Write}) \)
  - may happen in parallel w.r.t. barriers \( (p_1 \parallel p_2) \)

\( \Rightarrow (s_1, s_2) \) is a potential data race pair
Phase 1: Checking for Conflicts

- Two accesses $e_1 = (m_1, t_1, L_1, a_1, p_1, s_1)$ and $e_2 = (m_2, t_2, L_2, a_2, p_2, s_2)$ are in conflict when
  - memory range overlaps \( (m_1 \cap m_2 \neq \emptyset) \)
  - accesses from different threads \( (t_1 \neq t_2) \)
  - no common locks held \( (L_1 \cap L_2 = \emptyset) \)
  - at least one write \( (a_1 = \text{Write} \lor a_2 = \text{Write}) \)
  - may happen in parallel w.r.t. barriers \( (p_1 \parallel p_2) \)

\[ \Rightarrow (s_1, s_2) \text{ is a potential data race pair} \]

- How to efficiently check for conflicts?
  - Use appropriate data structures
Phase 1: $m_1 \cap m_2 \neq \emptyset$?

- **Skip Lists [Pugh90]**
  - A probabilistic alternative to balanced trees
  - Simple to understand and implement
  - Linked list augmented with extra levels of forward links

- **Interval Skip Lists [Hanson and Johnson ’92]**
  - Endpoints are kept in skip list
  - Edges and nodes are marked with intervals that contain them
  - Search for overlapping points and intervals in $O(\log n + K)$

![Diagram of Skip Lists and Interval Skip Lists](image)
Phase 1: $L_1 \cap L_2 = \emptyset$?

- **Lock tries**
  - Efficient search of racing accesses
  - No need to traverse edges in lock-set

Conflicts with $m = [3, 6)$
$L = \{ L_2, L_3 \}$
...

[L1, L4]

[1, 11)
Phase 1: $p_1 \parallel p_2$?

- May-happen-in-parallel analysis
  - Only search for accesses in a concurrent barrier phase

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier</td>
<td>barrier</td>
</tr>
<tr>
<td>barrier</td>
<td>barrier</td>
</tr>
<tr>
<td>access</td>
<td>access</td>
</tr>
<tr>
<td>barrier</td>
<td>barrier</td>
</tr>
<tr>
<td>barrier</td>
<td>barrier</td>
</tr>
</tbody>
</table>

Shared access between barriers

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>access</td>
<td>access</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
</tbody>
</table>

Shared access after wait

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
<tr>
<td>notify</td>
<td>wait</td>
</tr>
</tbody>
</table>

Shared access after notify
Phase 1: Checking for Conflicts

- Two accesses $e_1 = (m_1, t_1, L_1, a_1, p_1, s_1)$ and $e_2 = (m_2, t_2, L_2, a_2, p_2, s_2)$ are in conflict when
  - $m_1 \cap m_2 \neq \emptyset$: memory range overlaps
  - $t_1 \neq t_2$: accesses from different threads
  - $L_1 \cap L_2 = \emptyset$: no common locks held
  - $a_1 = \text{Write} \lor a_2 = \text{Write}$: at least one write
  - $p_1 \parallel p_2$: may happen in parallel w.r.t. barriers

  $\Rightarrow (s_1, s_2)$ is a potential data race pair

- How to efficiently check for conflicts?
  - Use appropriate data structures

- When do we check for conflicts?
  - Offline? Database becomes too large for long running programs
  - Online? Large overhead if checked at every memory access
Phase 1: Distributed Checking

- Minimize interaction between threads
  - Store shared memory accesses locally
  - At barrier boundary, send access information to respective owner of memory

- Conflict checking distributed among threads
Phase 1: Minimize Communication

- (Extended) Weaker-than relation
  - Only keep the *least protected* accesses
  - Prune provably redundant accesses [Choi et al ’02]
  - Also reduces superfluous race reports

- $e_1 \sqsubseteq e_2$ (access $e_1$ is weaker-than $e_2$) iff
  - larger memory range \( (e_1.m \supseteq e_2.m) \)
  - accessed by more threads \( (e_1.t = * \lor e_1.t = e_2.t) \)
  - smaller lockset \( (e_1.L \subseteq e_2.L) \)
  - weaker access type \( (e_1.a = \text{Write} \lor e_1.a = e_2.a) \)

- Lock trie allows efficient search for accesses weaker than $e$
  - Only traverse lock edges in $e.L$
Phase 1: Summary

- **Store shared memory access information locally**
  - Using efficient data structures (Interval Skip List and Lock Trie)
  - Keep only the weakest accesses

- **At barrier boundary, send access info to “owner” thread**

- **For each access e from other threads,**
  - Find overlapping intervals in IS-List
  - For each interval,
    - Check for existing weaker accesses (ignore e if found)
    - Check for racing accesses (report if found)
    - Add e to IS-List, and remove stronger accesses
Phase 2: Race Tester

- Consider each race pair from phase 1, say \((s_1, s_2)\)
- Control the scheduler as follows for each access:
  - Before \(T_1\) executes \(s_1\), block until some other thread executes \(s_2\)
  - Race pair associated with BUPC semaphore to block execution
  - When a matching \(s_2\) has overlapping address, report real race
- Two separate program runs per race pair \((s_1->s_2, s_2->s_1)\)
  - Testing controlled by script – 2N runs where N is number of races predicted in phase 1
Phase 2: Race Tester

- Consider each race pair from phase 1, say \((s_1, s_2)\)
- Control the scheduler as follows for each access:
  - Before \(T_1\) executes \(s_1\), block until some other thread executes \(s_2\)
  - Race pair associated with BUPC semaphore to block execution
  - When a matching \(s_2\) has same address, report real race
- Two separate program runs per race pair (\(s_1 \rightarrow s_2\), \(s_2 \rightarrow s_1\))
- Testing controlled by script – 2N runs where N is number of races predicted in phase 1

To ensure progress,
1) Block with a timeout
2) Block while avoiding deadlock
Phase 2: Example

\[ (a:\text{Write}, p:1, s:6), (a:\text{Read}, p:1, s:5) \]

T1 → T2

5: Read 0x8000

Paused: \{\}
Phase 2: Example

\[
\langle (a:\text{Write, } p:1, s:6), (a:\text{Read, } p:1, s:5) \rangle
\]

T1 \quad T2

5: Read 0x8000

Paused: \{\}
Phase 2: Example

Paused: \{(m:0x8000, a:Read, p:1, s:5)\}

\langle (a:Write, p:1, s:6), (a:Read, p:1, s:5) \rangle

T1

T2

5: Read 0x8000
Phase 2: Example

Paused: \{m:0x8000, a:Read, p:1, s:5\}

\<(a:Write, p:1, s:6), (a:Read, p:1, s:5)\>
Phase 2: Example

\[
\langle a: \text{Write}, p:1, s:6 \rangle, \ (a: \text{Read}, p:1, s:5) >
\]

Paused: \{ (m:0x8000, a: \text{Read}, p:1, s:5) \}
Phase 2: Example

Paused: \{(m:0x8000, a:Read, p:1, s:5)\}

\(<(a:\text{Write, p:1, s:6}), (a:\text{Read, p:1, s:5})>\)
Phase 2: Example

\[ <(a:\text{Write}, p:1, s:6), (a:\text{Read}, p:1, s:5)> \]

Paused: \{(m:0x8000, a:\text{Read}, p:1, s:5)\}

T1
5: Read 0x8000
5: Read 0x9000
5: Read 0xA000

T2
5: Read 0x8000
Phase 2: Example

Paused: \{ (m:0x8000, a:Read, p:1, s:5) \}
Phase 2: Example

Paused: \{(m:0x8000, a:Read, p:1, s:5)\}
Phase 2: Example

\[
\langle (a: Write, p: 1, s: 6), (a: Read, p: 1, s: 5) \rangle
\]

- T1:
  5: Read 0x8000
  5: Read 0x9000
  5: Read 0xA000
  5: Read 0xB000
  6: Write 0x8000

- T2:
  5: Read 0x8000

Paused: \{ (m: 0x8000, a: Read, p: 1, s: 5) \}
Phase 2: Example

Paused: {\((m:0x8000, a:Write, p:1, s:6), (m:0x8000, a:Read, p:1, s:5)\)}
Phase 2: Example

\[
<(a:Write, p:1, s:6), (a:Read, p:1, s:5)>
\]

5: Read 0x8000
5: Read 0x9000
5: Read 0xA000
5: Read 0xB000
6: Write 0x8000

s5 then s6
Phase 2: Example

<(a:Write, p:1, s:6), (a:Read, p:1, s:5)>

T1

5: Read 0x8000
5: Read 0x9000
5: Read 0xA000
5: Read 0xB000
6: Write 0x8000

T2

5: Read 0x8000

s6 then s5: Non-deterministic result!
## Results

- Test results for single node (4 threads)
  - Intel Core i7 2.66 GHz (quad core) with 8GB RAM
  - Low overhead for race detection
  - Detects and confirms real races in benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normal Runtime</th>
<th>Race detector (phase 1)</th>
<th>Race tester (phase 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime</td>
<td>Pot. races</td>
<td>Runtime</td>
</tr>
<tr>
<td>guppie</td>
<td>2.08 sec</td>
<td>2.30 sec</td>
<td>2 (157)</td>
</tr>
<tr>
<td>laplace</td>
<td>2.08 sec</td>
<td>2.34 sec</td>
<td>0</td>
</tr>
<tr>
<td>mcop</td>
<td>2.16 sec</td>
<td>2.17 sec</td>
<td>0</td>
</tr>
<tr>
<td>psearch</td>
<td>3.06 sec</td>
<td>3.06 sec</td>
<td>2 (73)</td>
</tr>
</tbody>
</table>
Results

- Performance results compared to previous version
  - Optimizations helpful, especially for memory intensive benchmark
Bugs Found

- In NPB 2.3 FT,
  - Wrong lock allocation function causes real races in validation code
  - Spurious validation failure errors

```c
shared  dcomplex *dbg_sum;
static  upc_lock_t  *sum_write;

sum_write = upc_global_lock_alloc();  // wrong function

upc_lock (sum_write);
{
    dbg_sum->real = dbg_sum->real + chk.real;
    dbg_sum->imag = dbg_sum->imag + chk.imag;
}
upc_unlock (sum_write);
```
Bugs Found

- In SPLASH2 lu,
  - Multiple initialization of vector without locks
  - Benign but performance bug

```c
void InitA()
{
    ...
    for (j=0; j<n; j++) {
        for (i=0; i<n; i++) {
            rhs[i] += a[i+j*n]; // executed by all threads
        }
    }
}
```
Roadmap

- Comprehensive instrumentation to handle all types of shared memory accesses
  - Handling local aliases to shared heap
    - Requires compiler instrumentation
  - Non-blocking reads and writes
    - Match syncs and inits with correct thread scheduling semantics
  - Handling collectives

- Additional analyses
  - Memory consistency analysis
  - Deadlock analysis
  - Approximate record and replay

- Make scalable on small cluster based environments
Conclusion

- Active testing has been successfully used to find and reproduce real bugs in Java and C/C++ programs
- Preliminary implementation for UPC
  - Efficient data race detector and tester
- Comprehensive analysis and testing framework for UPC in progress

http://upc.lbl.gov/thrille.shtml