Avoidance, Detection, and Repair of Bugs in Structured Parallel Programs

Vivek Sarkar
E.D. Butcher Chair in Engineering
Professor of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments --- Habanero Extreme Scale Software Group

- **Faculty**
  - Vivek Sarkar

- **Senior Research Scientists**
  - Michael Burke, Kathleen Knobe

- **Research Scientists**
  - Zoran Budimlić, Philippe Charles, Michael Fagan, Akihiro Hayashi, Vivek Kumar, Jun Shirako, Jisheng Zhao

- **Postdoctoral Researcher**
  - Tiago Cogumbreiro

- **Post-MS PhD Students**
  - Kumud Bhandari, Max Grossman, Alina Sbirlea, Rishi Surendran, Sağnak Taşırılar, Nick Vrvilo

- **Pre-MS PhD Students**
  - Prasanth Chatarasi, Arghya Chatterjee, Ankush Mandal, Yuhan Peng, Jonathan Sharman
With Multicore Processors and Cloud Computing, all Computers are Parallel Computers …

- Datacenter: $10^9$ threads
- Rack: $10^4$-$10^5$ threads
- Socket/blade: 500-5000 threads
- Die: 100-1000 threads
- Core/tile: 1-10 threads
... and all Software is Parallel by Default!

- New classes of bugs are being encountered in new programming models and frameworks across the full spectrum of parallel systems (embedded, mobile, server, cloud)

- New challenges for software correctness and reliability
  
  A. **Avoidance** of parallelism/concurrency bugs
  
  B. **Detection** of parallelism/concurrency bugs
  
  C. **Repair** of parallelism/concurrency bugs
Two-level programming model
Declarative Coordination Language for Domain Experts:
CnC, DFGL
+
Task-Parallel Languages for Parallelism-aware Developers:
Habanero-C, Habanero-C++, Habanero-Java, Habanero-Scala

Structured-parallel execution model
1) Lightweight asynchronous tasks and data transfers
   - Creation: \textit{async tasks, future tasks, data-driven tasks}
   - Termination: \textit{finish, future get, await}
   - Data Transfers: \textit{asyncPut, asyncGet}
2) Locality control for task and data distribution
   - Computation and Data Distributions: \textit{hierarchical places, global name space}
3) Inter-task synchronization operations
   - Mutual exclusion: \textit{isolated, actors}
   - Collective and point-to-point operations: \textit{phasers, accumulators}

http://habanero.rice.edu
Our Approach: Leverage Structured Parallelism

- Programming models should specify what can run in parallel, not how the parallelism should be exploited
  - Specify logical (rather than actual) parallelism with "structured primitives that are accompanied by strong semantic guarantees"
- Compilers should be able to analyze and transform parallel programs
  - Extend foundations of compiler theory so as to analyze and transform structured parallel programs
- Runtime systems should be able to efficiently manage larger degrees of parallelism than the underlying hardware
  - Build scalable and adaptive runtime systems for structured parallelism that trade off parallelism, locality, energy, and resilience
- Debugging and verification tools should be sound and complete, to the largest extent possible
  - Use structured parallel abstractions to help programmers avoid, detect and repair bugs in parallel programs
Structured Primitives in Habanero Execution Model

1) Lightweight asynchronous tasks and data transfers
   - Creation: async tasks, future tasks, data-driven tasks
   - Termination: finish, future get, await
   - Data Transfers: asyncPut, asyncGet

2) Locality control for control and data distribution
   - Computation and Data Distributions: hierarchical places, global name space

3) Inter-task synchronization operations
   - Mutual exclusion: global/object-based isolation, actors
   - Collective and point-to-point operations: phasers, accumulators

Note: these primitives can be used directly as a programming model, or can be targeted by higher level programming models
Properties of interest:
- DLF = DeadLock-Free
- DRF = Data-Race-Free
- DET = Structural + Functional Determinism
- DRF $\Rightarrow$ DET = DRF implies DET
- SER = Serial elision

If a Habanero program only uses async, finish, and final future constructs, then it is guaranteed to belong to the SER + DLF + (DRF $\Rightarrow$ DET) class

Adding phasers yields programs in the DLF + (DRF $\Rightarrow$ DET) class (dropping SER)

Adding async await yields programs in the DRF $\Rightarrow$ DET class (dropping DRF)

Restricting shared data accesses to futures, isolated, actors yields programs in the DRF-ALL class

...
Part A: Overall Approach to Bug Avoidance

- Establish sufficient conditions to ensure that bug cannot appear in any execution of any program that satisfies those conditions

- Example: Deadlock Avoidance
Deadlock Avoidance in Unstructured Fork-Join is hard

It can be hard to avoid deadlocks with unstructured parallelism, e.g.,

1. `static Thread t1, t2;`
2. `t1 = new Thread(() -> {t2.join();});`
3. `t2 = new Thread(() -> {t1.join();});`
4. `t1.start();`
5. `t2.start();`
Deadlock Avoidance can be guaranteed for Structured Fork-Join parallelism (async-finish, spawn-sync, …)

```
finish {
  Task A0 (Part 1);
  async {A1; async A2;}
  try {
    finish {
      Task A0 (Part 2);
      async A3;
      async A4;
    }
    catch (...) { ... }
    Task A0 (Part 3);
  }
}
```
Barriers: another example of deadlock (or undefined behavior) with unstructured parallelism

1. // Assume that number of threads is >= 2
2. #pragma omp parallel
3. {
4.     const int tid = omp_get_thread_num();
5.     if (tid != 1) {
6.         #pragma omp barrier
7.     }
8. }

Non-conforming program leads to unpredictable results on different platforms

Deadlock, silent completion, …

Similar examples can be created for other models, e.g., MPI
Phasers: a structured generalization of barriers and point-to-point synchronization

- **Phaser allocation:** `phaser ph = new phaser(mode);`
  - Phaser `ph` is allocated with registration `mode`.
  - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF).
  - Registration mode lattice:
    - `SINGLE`:
      - `SIG_WAIT` (default)
      - `SIGNAL` → `WAIT`

- **Task creation:** `async phased (ph₁<mode₁>, ph₂<mode₂>, … ) <stmt>`
  - Spawned task is registered with `ph₁` in `mode₁`, `ph₂` in `mode₂`, …
  - Child task’s capabilities must be subset of parent’s.
  - *Task drops all phaser registrations upon termination*.

- **Synchronization:** `next;`
  - Advance each phaser that activity is registered on to its next phase.
  - Semantics depends on registration mode.
Deadlock avoidance is guaranteed with phasers ...

```
finish {
    phaser ph = new phaser();  //A₁
    async phased(ph){    STMT1; next; STMT2; next; STMT3; }  //A₂
    async phased(ph){    STMT4: next; STMT5; }  //A₃
    STMT6; next; STMT7; next; STMT8;  //A₁
}
```

Tasks $A_1$, $A_2$, $A_3$ are registered on phaser ph (can be extended with signal/wait modes)

Dynamic parallelism: 
# activities registered on phaser can vary
... even with point-to-point synchronization

1. finish for (point[i]: [1:N])
2. async phased(ph[i]<SIG>, ph[i-1]<WAIT>,
   ph[i+1]<WAIT>) {
3.   while ( true ) {
4.     A[i] = F(B[i-1], B[i], B[i+1]);
5.     next; // barrier
6.     if ( equals(A[i],B[i]) ) break;
7.     else B[i] = A[i];
8.   } // while
9. } // finish-for-async

Exiting from while loop terminates
for-async iteration i, and
automatically “deregisters” task i
from its phasers

Deadlock avoidance proof
formalized in Coq
Futures can deadlock if their references participate in a data race ...

```java
future<int> f1=null;
future<int> f2=null;

void main(String[] args) {
    f1 = async<int> {return a1();};
f2 = async<int> {return a2();};
    ...
}

int a1() {
    future<int> tmp=null;
    do {
        tmp=f2;
    } while (tmp == null);
    return tmp.get();
}

int a2() {
    future<int> tmp=null;
    do {
        tmp=f1;
    } while (tmp == null);
    return tmp.get();
}
```

... a sufficient condition to guarantee deadlock avoidance with futures is to ensure that all future references are declared as final variables
Part B: Overall Approach to Bug Detection

- For bugs that are not guaranteed to be avoided, we need to turn to detection.
- Focus of our work is on dynamic bug detection for soundness and precision, supported by static analysis for efficiency.

Examples:
1. Data Race Detection
2. Permission Violation Detection
3. Commutativity Violation Detection
Two accesses to a shared memory location by two different tasks result in a data race if:

- At least one of the access is a write, and
- The program structure imposes no happens-before ordering between the two accesses

This definition is sometimes referred to as a potential data race.
SPD3: Scalable and Precise Dynamic Datarace Detection algorithm

- A parallel sound and precise race detection algorithm for async and finish constructs
- Two components:
  - Dynamic Program Structure Tree (DPST)
    - To identify potentially parallel accesses
  - Access Summary
    - To identify interfering accesses
- “Scalable and Precise Dynamic Data Race Detection for Structured Parallelism”. Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, Eran Yahav. [PLDI ‘12]
Dynamic Program Structure Tree (DPST)

- Tree that maintains parent-child relationships among async, finish, and step instances
  - Internal nodes represent async and finish instances
  - Leaf nodes represent step instances

- Step
  - Maximal sequence of statements with no async or finish

- Children of a node are ordered from left-to-right
  - Reflects the sequencing of computations that belong to the same task
DPST Example

1: `finish` { // F1
2:     S1;
3:     `async` { // A1
4:         `async` { // A2
5:             S2;
6:         }
7:     `async` { // A3
8:         S3;
9:     }
10:    S4;
11: }
12:    S5;
13:    `async` { // A4
14:        S6;
15: }
16: }

Left-to-right ordering of children
DPST Properties resulting from Structured Parallelism

- Every execution of a program with the same input produces the same DPST
  - If no data race is detected

- Path from a leaf to the root stays invariant as the tree grows

- All computations happen in leaves
  - May-happen-in-parallel checks will be done only between leaves
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L := \text{LCA} (S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
Else return false

Assuming \(S\) is to the left of \(S'\) in the DPST
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L := \text{LCA} (S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async return true
   Else return false
Identifying Parallel Accesses using DPST

**DMHP (S, S’)**

1) \( L := \text{LCA} (S, S’) \)
2) \( C := \text{child of} \ L \text{ that is ancestor of} \ S \)
3) If \( C \) is async
   return true
Else return false
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L := \text{LCA}(S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP \((S, S')\):

1) \(L := \text{LCA} \ (S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
Else return false
Identifying Parallel Accesses using DPST

**DMHP (S, S')**

1) \( L := \text{LCA} (S, S') \)
2) \( C := \text{child of } L \text{ that is ancestor of } S \)
3) If \( C \) is async
   return true
   Else return false

A1 is an async => DMHP(S3, S6) = true
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L := \text{LCA} (S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
Else return false
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L \coloneqq \text{LCA} (S, S')\)
2) \(C \coloneqq \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
Else return false
Identifying Parallel Accesses using DPST

**DMHP \((S, S')\)**

1) \(L := \text{LCA} (S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async
   return true
   Else return false
Identifying Parallel Accesses using DPST

DMHP \((S, S')\)

1) \(L := \text{LCA}\) \((S, S')\)
2) \(C := \text{child of } L \text{ that is ancestor of } S\)
3) If \(C\) is async return true
   Else return false

\(\text{S5 is NOT an async } \Rightarrow \text{DMHP(S5, S6) = false}\)
## Related Work: A Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Language</td>
<td>Nested Fork-Join &amp; Synchronization operations</td>
<td>Nested Fork-Join</td>
<td>Spawn-Sync</td>
<td>Spawn-Sync</td>
<td>Unstructured Fork-Join</td>
<td>Async-Finish</td>
<td>Async-Finish</td>
</tr>
<tr>
<td>Space Overhead per memory location</td>
<td>$O(m)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Guarantees</td>
<td>Per-Schedule</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
<td>Per-Input</td>
</tr>
<tr>
<td>Empirical Evaluation</td>
<td>No</td>
<td>Minimal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Execute Program in Parallel</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dependent on Scheduling technique</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**OTFDAA** – On the fly detection of access anomalies

$m$ – number of threads executing the program

$N$ – maximum logical concurrency in the program
Another Example: Detection of Permission Violations

- Permissions check for “high-level” data races
- Advances in Permission Types:
  - Aliased write permissions
  - Dynamic permission acquires/releases
  - Storable permissions
- Extensions:
  - Array-Based Parallelism
  - Object-based isolation
Permission Types in Code

```java
write void insert (write Node n) {
    n.next = next;
    next = n;
}

read bool search (int i) {
    if (data == i)
        return true;
    else if (next == null)
        return false;
    else return next.search (i);
}
```
void insert (Node n) {
    n.next = next; next = n;
}

bool search (int i) {
    if (data == i) return true;
    else if (next == null) return false;
    else return next.search (i);
}
Acquires & Fail-Stop Semantics

Permission violations are bugs! (Like null pointer dereferences)

Dynamic Permission Conflict

Exception

Block

- Changes synchronization behavior
- Could cause deadlock
Object Modes

- Shared Read-Only
- Private Read-Write
Object Modes

Shared Read-Only → Private Read-Write (task 1) → Private Read-Write (task 2) → ...

RICE
Fractional Permissions

- **Exclusive permission**
- **Shared read permission**
Gradual Typing enables Trade-off between User Effort and Dynamic Checks

- ~1.5x (geo mean) [RV’11]
- < 5% (average) [ECOOP’12]

Slowdown

Productivity (LoC modified)
Dynamic Determinism Checking for Structured Parallelism [WoDet’14]

• HJd = Habanero Java with determinism
  – Builds on our prior race-freedom work [RV’11, ECOOP’12]
• Determinism is checked dynamically
  – For application code, not parallel libraries
• Determinism failures throw exceptions
  – Because non-determinism is a bug!
• Checking itself uses a deterministic structure
• Leads to low overhead: 1.26x slowdown!
Two Sorts of Code

1. High-performance parallel libraries
   – Uses complex and subtle parallel constructs
   – Written by concurrency experts: the 1%

2. Deterministic application code
   – Uses parallel libraries in a deterministic way
   – Parallelism behavior is straightforward
   – Written by everybody else: the 99%

We focus on application code
Approach: Determinism via Commutativity

1. Identify pairs of library operations which commute
   - Operations = parallel library primitives (the 1%)
   - Verified independently of this work

2. Dynamic checking of the application code (the 99%)
   - Detect commutativity violations using the DPST
   - Ensures no non-commuting methods could possibly run in parallel
Example: Counting Factors in Parallel

class CountFactors {
    int countFactors (int n) {
        AtomicInteger cnt = new AtomicInteger();
        finish {
            for (int i = 2; i < n; ++i)
                async {
                    if (n % i == 0)
                        cnt.increment();
                }
            return cnt.get();
        }
    }
}
Specifying Commutativity for Libraries

• Methods annotated with “commutativity sets”
  – Each pair of methods in set commute
• Syntax:
  \[ \text{@CommSets}\{S_1, \ldots, S_n\} \text{ <method sig}> \]
  – States method is in sets \( S_1 \) through \( S_n \)
  – Commutes with all other methods in these sets
Commutativity Sets for AtomicInteger

```java
final class AtomicInteger {
    @CommSets{"read"} int get () { ... }
    @CommSets{"modify"} void increment()
        { ... }
    @CommSets{"modify"} void decrement()
        { ... }
    @CommSets{"read","modify"} int initValue()
        { ... }
    int incrementAndGet () { ... }
}
```

- `get` commutes with itself
- Inc/Dec commute with themselves and each other
- Commutes with anything
- Commutes with nothing (not even itself)
Part C: Test-Driven Repair of Data Races

- Use test inputs to drive program repair by inserting finish statements to ensure that no races remain for the test inputs
- Goal: maximize available parallelism after repair
- The newly inserted finish statements must respect the lexical scope of the draft program
- The complete program after insertion of finish statements must have the same semantics as its linearized version (eliding parallel constructs)
Parallel Software Development: Current Practice

Sequential Program
Parallel Software Development: Current Practice

Sequential Program

Program with parallelism
Parallel Software Development: Current Practice

Sequential Program

Program with parallelism

Results don’t match. Let me try adding synchronization.
Parallel Software Development: Current Practice

Sequential Program

Poor Parallel Performance

Program with parallelism

Program with data race

Maybe I can remove some synchronization!
Parallel Software Development: Our Vision

Sequential Program

Complete race-free high-performance parallel program

Draft program with parallelism but no synchronization

↑..↑
Test inputs
High Level View of Test-Driven Program Repair

Program → Test input 1 → Test input 2 → ... → Test input n → Repaired Program
Tool guarantees data race freedom in repaired program for all test inputs
Overview of Our Approach

- Extended ESP-Bags data race detector
  - Performs a sequential depth first execution of the program on a single processor
- Dynamic finish placement finds an optimal solution
- Static finish placement finds a heuristic solution
Coupling Between Static and Dynamic Finish Placement

Dynamic Finish Placement

Static Finish Placement

```java
public static void main (...)
{
  ... S1; ... 
  ...
}
```
Coupling Between Static and Dynamic Finish Placement

Dynamic Finish Placement

Static Finish Placement

```java
public static void main (...) {
  ...
  finish { S1; ...}
  ...
}
```
Coupling Between Static and Dynamic Finish Placement

Dynamic Finish Placement

Static Finish Placement

```
public static void main (...) {
  ...
  finish { S1; ...}
  ...
}
```
Program Repair Example: Quicksort

```java
1 static void quicksort(int[] A, int M, int N) {
2   if (M < N) {
3       point p = partition(A, M, N);
4       int I = p.get(0);
5       int J = p.get(1);
6       async quicksort(A, M, J);
7       async quicksort(A, I, N);
8   }
9 }
10 ...
11 quicksort(A, 0, size-1); //Call inside main
12 /* verify results */
```

Input program has data races
Program Repair Example: Quicksort

1 static void quicksort(int[] A, int M, int N) {
2   if(M < N) {
3     point p = partition(A, M, N);
4     int I = p.get(0);
5     int J = p.get(1);
6     async quicksort(A, M, J);
7     async quicksort(A, I, N);
8   }
9 }
10 ...  
11 quicksort(A, 0, size-1); //Call inside main
12 /* verify results */

Input program has data races
Program Repair Example: Quicksort

```java
static void quicksort(int[] A, int M, int N) {
    if(M < N) {
        point p = partition(A, M, N);
        int I = p.get(0);
        int J = p.get(1);
    async quicksort(A, M, J);
    async quicksort(A, I, N);
    }
}
```

Too much synchronization
Program Repair Example: Quicksort

static void quicksort(int[] A, int M, int N) {
  if(M < N) {
    point p = partition(A, M, N);
    int I = p.get(0);
    int J = p.get(1);

    async quicksort(A, M, J);
    async quicksort(A, I, N);
  }
}

quicksort(A, 0, size-1); //Call inside main
/* verify results */

Too much synchronization
Program Repair Example: Quicksort

```java
1 static void quicksort(int[] A, int M, int N) {
2     if(M < N) {
3         point p = partition(A, M, N);
4         int I = p.get(0);
5         int J = p.get(1);
6         async quicksort(A, M, J);
7         async quicksort(A, I, N);
8     }
9 }
10 ... 
11 quicksort(A, 0, size-1); //Call inside main
12 /* verify results */
```

Best finish placement
Student Homework Evaluation

• Evaluated student homework submissions as part of an undergraduate course on parallel computing

• Week 1 Assignment: Perform manual repair of buggy quicksort program with missing finish constructs

• Compared 59 student submissions against the repair performed by the tool
  - 5 submissions had data races
  - 29 submissions were over-synchronized
  - 25 submissions matched the output from repair tool
Other Related Topics

- Determinism checking [SAS ‘10, WoDet ‘14]
- Deterministic reductions [WoDet ‘11, WoDet ‘13]
- Definitions of Functional vs. Structural Determinism, Determinacy, Repeatability [DFM ‘12]
- Delegated Isolation for Nested Task Parallelism [OOPSLA ‘11, OOPSLA ‘13]
- Object-based Isolation [EuroPar ‘15]
- Integrating Actors with Task Parallelism [OOPSLA ‘12, AGERE ‘14]
- Model Checking Task Parallel Programs using Gradual Permissions [ASE ‘15]
- Analysis and Transformation of Parallel Programs [TOPLAS ‘13, LCPC ‘15, PACT ’15]
- See Publications link in http://habanero.rice.edu
Conclusions

- New challenges for correctness and reliability in parallel software
  - Avoidance of parallelism/concurrency bugs
  - Detection of parallelism/concurrency bugs
  - Repair of parallelism/concurrency bugs
- Structured-parallel primitives can provide foundation for addressing these challenges
- This talk presented early experiences from the Habanero project, and key structured-parallel primitives that can enable effective avoidance, detection, and repair of parallel bugs