Efficient Data Race Detection for Async-Finish Parallelism

Raghavan Raman Jisheng Zhao Vivek Sarkar
Rice University

Martin Vechev Eran Yahav
IBM T. J. Watson Research Center

Structured Parallelism

- Many algorithms expressible in structured parallel languages
  - Renewed interest: Java fork-join, MIT's Cilk, IBM's X10, DPJ (UIUC), TPL (Microsoft), HJ (Rice)

- Benefits of structured languages
  - Eliminates certain kinds of deadlocks
  - Enables simpler analysis of concurrency
  - Amenable to schedulers with guaranteed space and time bounds

Structured Parallelism

- Programs contain a massive number of tasks
  - Most tasks access disjoint memory locations

- Current data-race detection techniques
  - Focus on unstructured lock-based concurrency
  - Poor space complexity
    - Size of a Vector Clock is proportional to number of threads
  - Inefficient when applied to huge number of tasks

Structured Parallelism: Cilk

- Cilk language constructs
  - spawn
  - sync
  - Induces fully-strict computation graphs (also called Series-Parallel Dags)

- SP-bags: Cilk data-race detection algorithm
  - Key idea: Exploits structure for checking
  - Good space complexity – O(1) per memory location, independent of number of tasks
  - Serial algorithm – runs on a single worker thread but reports all possible data races for a given input

SP-bags Revisited

- SP-bags: targets spawn-sync
  - Limitation: Not directly applicable to other constructs
- async-finish
  - More general set of computation graphs than spawn-sync
  - Used as a basis in research projects like IBM X10, Rice HJ, UCLA FX10
- Can SP-bags be extended to async-finish ?

Main Contributions

- ESP-bags algorithm: Extending SP-bags to async-finish
- Implementation in tool - TaskChecker
- Static compiler optimizations to reduce runtime overhead
- Evaluation on 12 benchmarks
  - Average slowdown is 3.05x (Geometric Mean)
    - Average slowdown for SP-bags is 7.05x
    - Average slowdown for FastTrack is 6.19x

Task Parallel Extensions

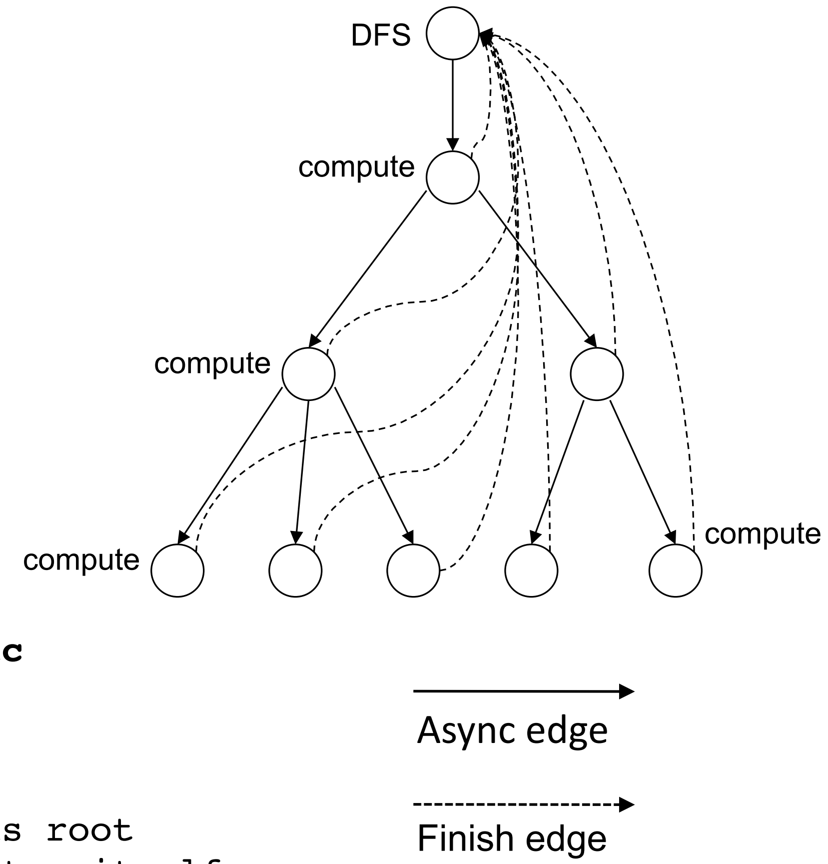
- async <stmt>:
  - Creates a new task that can execute <stmt> in parallel with the parent task
- finish <stmt>:
  - Blocks and waits for all tasks spawned inside finish scope to complete
- isolated <stmt>:
  - Each task must perform an isolated statement in mutual exclusion with any other isolated statements
  - Also called as `atomic' in X10

Example – Parallel Depth-First Search Spanning Tree

```
class V  {
  V [] neighbors;
  V parent;
  . . .
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent = n;
    return parent == n;
  } // tryLabeling

  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
        async child.compute(); //escaping async
    }
  } // compute

  void DFS() { // Compute a DFST with "this" as root
     parent = this; // Only the root has parent = itself
     finish compute();
  } // DFS
} // class V
```



ESP-bags: Extended SP-bags

- Attach two ‘bags’, S and P, to every task instance

- Also attach a P-bag to every finish instance
  - Different from SP-bags

- Each bag holds a set of task ids
  - Task ids are created dynamically

- Attach meta-data to memory locations
  - Each memory location has two fields:
    - reader task id
    - writer task id
  - Can be restricted to two fields per object for object-based race-detection

ESP-bags: Basic Operation

- A serial algorithm:
  - Performs a sequential depth-first execution of the parallel program on a single processor
- Invariant:
  - A task id will always belong to at most one bag at a time
- Space Overhead:
  - Bags represented using a disjoint-set data structure

Contents of S and P bags

- When a statement S in task A is executed, …
- S-bag of task A
  - holds the task ids of descendent tasks (of A) that always precede the statement S in A
- P-bag of task A
  - holds the task ids of descendent tasks (of A) that may execute in parallel with the statement S in A

Example

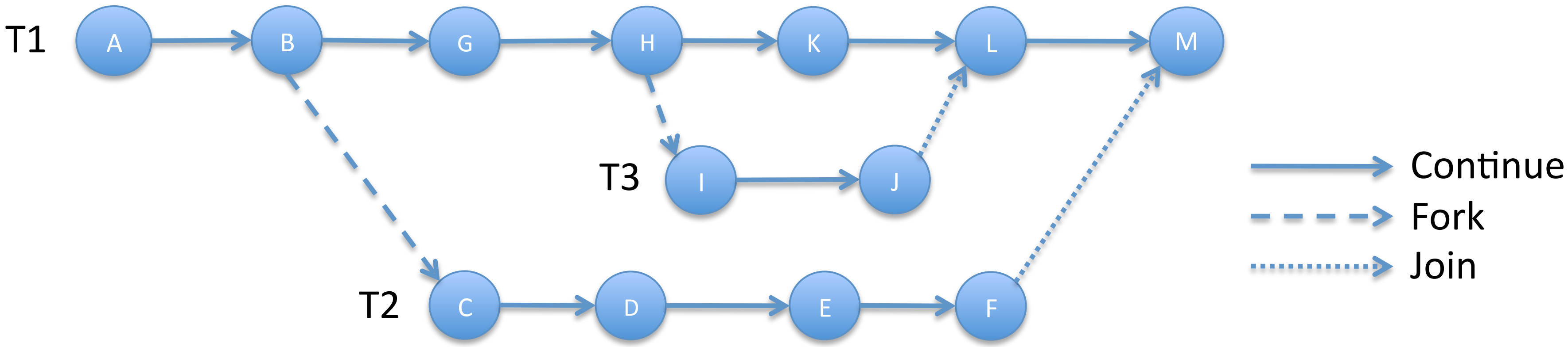
S_{T1} = { T1 }
P_{T1} = { }

S_{T1} = { T1 }
P_{T1} = { T2 }

S_{T1} = { T1 }
P_{T1} = { T2, T3 }

S_{T1} = { T1, T3 }
P_{T1} = { T2 }

S_{T1} = { T1, T2, T3 }
P_{T1} = { }



Updating the Bags

- Creation of task A (async A)
  - $S_A = \{ A \}$
  - $P_A = \{ \}$

- Execution returns from task A to parent task/finish B
  - $P_B = P_B \cup S_A \cup P_A$
  - $S_A = \{ \}$
  - $P_A = \{ \}$

- Start of finish block F
  - $P_F = \{ \}$

- End of finish block F by task B
  - $S_B = S_B \cup P_F$
  - $P_F = \{ \}$

Checks Performed When Accessing the memory locations

- Read location L by task t
  - If L.writer is in a P-bag, then Data Race
  - If L.reader is in a S-bag, then L.reader = t
  - Otherwise, do nothing

- Write Location L by Task t
  - If L.writer or L.reader is in a P-bag, then Data Race
  - L.writer = t

Example

```
   T1
1  final int [] A, B;
2  …
3  finish {  F1
4      for (int i = 0; i < size; i++) {
5          …
6          async {  T2
7              B[i] += i;
8              …
9          } // async
10         finish {  F2
11             async {  T3
12                 B[i] = A[i];
13             } // async
14             …
15         } // finish
16     } // for
17 } // finish
```

PC	T1 S	F1 P	T2 S	F2 P	T3 S	B[0] Writer
1	{ T1 }	-	-	-	-	-
3	{ T1 }	{ }	-	-	-	-
6	{ T1 }	{ }	{ T2 }	-	-	-
7	{ T1 }	{ }	{ T2 }	-	-	T2
9	{ T1 }	{ T2 }	{ }	-	-	T2
10	{ T1 }	{ T2 }	{ }	{ }	-	T2
11	{ T1 }	{ T2 }	{ }	{ }	{ T3 }	T2

Reduced version of the example in Figure 2 in the paper

Example

```
    T1
1  final int [] A, B;
2  …
3  finish {  F1
4      for (int i = 0; i < size; i++) {
5          …
6          async {  T2
7              B[i] += i;
8              …
9          } // async
10         finish {  F2
11             async {  T3
12                 B[i] = A[i];
13             } // async
14             …
15         } // finish
16     } // for
17 } // finish
```

PC	T1 S	F1 P	T2 S	F2 P	T3 S	B[0] Writer
1	{ T1 }	-	-	-	-	-
3	{ T1 }	{ }	-	-	-	-
6	{ T1 }	{ }	{ T2 }	-	-	-
7	{ T1 }	{ }	{ T2 }	-	-	T2
9	{ T1 }	{ T2 }	{ }	-	-	T2
10	{ T1 }	{ T2 }	{ }	{ }	-	T2
11	{ T1 }	{ T2 }	{ }	{ }	{ T3 }	T2
12	{ T1 }	{ T2 }	{ }	{ }	{ T3 }	T3

Data Race

Reduced version of the example in Figure 2 in the paper

Performance Optimizations

- ESP-bags Algorithm:
  - Instrument every memory access
  - Some checks are redundant and can be removed

- Read/Write Check Optimization

```
…
async {
   ReadCheck (p.x);
   … = p.x;
   …
   WriteCheck (p.x);
   p.x = …;
}
```

More Performance Optimizations

- Check Elimination in Sequential Code Regions
- Read-only Check Elimination in Parallel Regions
- Eliminating Checks on Task-local Objects
- Loop Invariant Check Optimization

Experimental Setup

- 16-way (4x4) Intel Xeon 2.4GHz system
  - 30 GB memory
  - Red Hat Linux (RHEL 5)
- Sun Hotspot JDK 1.6
- All benchmarks written in HJ using only Finish/Async/Isolated constructs
  - JGF benchmarks used with their highest input size
    - Except MolDyn for which size A was used
  - http://habanero.rice.edu/hj
- ESP-bags algorithm – implemented in a tool called TaskChecker, along with the optimizations

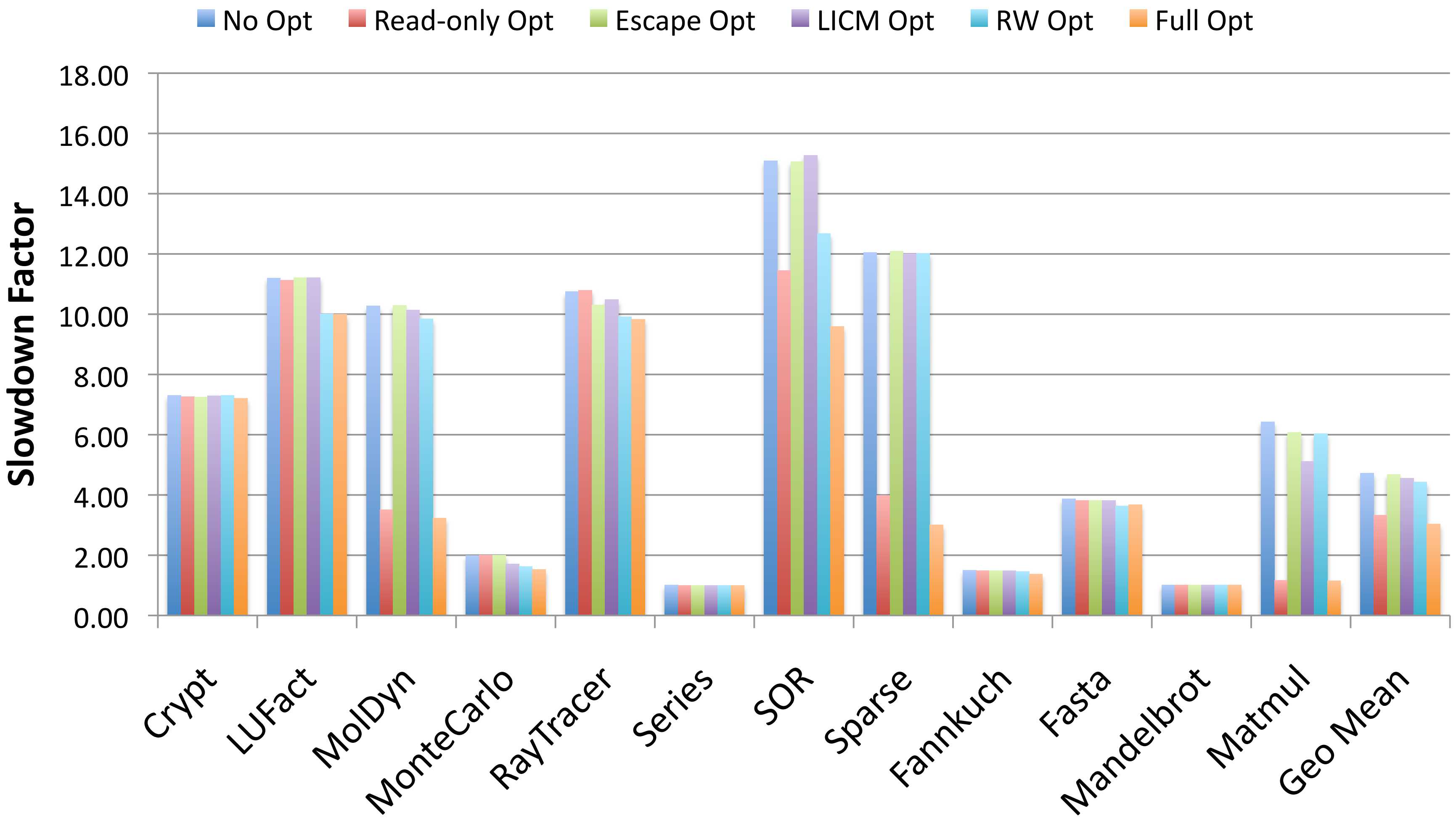
Slowdown of ESP-bags Algorithm

Benchmark	Number of Tasks	Original Time (s)	ESP-bags Slowdown Factor	
			w/o Opts	w/ Opts
Crypt	1.3e7	15.24	7.63	7.29
LUFact	1.6e6	15.19	12.45	10.08
MolDyn	5.1e5	45.88	10.57	3.93
MonteCarlo	3.0e5	19.55	1.99	1.57
RayTracer	5.0e2	38.85	11.89	9.48
Series	1.0e6	1395.81	1.01	1.00
SOR	2.0e5	3.03	14.99	9.05
SparseMatMult	6.4e1	13.59	12.79	2.73
Fannkuch	1.0e6	7.71	1.49	1.38
Fasta	4.0e0	1.39	3.88	3.73
Mandelbrot	1.6e1	11.89	1.02	1.02
Matmul	1.0e3	19.59	6.43	1.16
Geometric Mean			**4.86**	**3.05**

Comparison with Other Race Detectors

Properties	SP-bags	ESP-bags	FastTrack
Language	Cilk	X10/HJ	Java
Number of Test Programs	8	12	12
Minimum Slowdown	2.41	1	0.9
Maximum Slowdown	11.09	10.08	14.8
Average Slowdown (Geometric Mean)	7.05	3.05	6.19
Space Overhead	O(1)	O(1)	O(n)
Serial/Parallel	Serial	Serial	Parallel
Guarantees	Per-Input	Per-Input	Per-Execution
Schedule Dependent?	No	No	Yes

Breakdown of Optimizations



Extensions for Isolated

- Need to check that isolated and non-isolated accesses that may execute in parallel do not conflict
- Two additional fields
  - Can be restricted to memory locations accessed in isolated blocks
- Refer to the paper for more details

Determinism vs. Data Races

- Important for structured parallel algorithms
  - Intended to be deterministic
- Data-race freedom implies determinism in some cases (only with async-finish)
  - Implies temporal and spatial disjointness
- Algorithm dynamically checks for determinism
  - Guarantees per-input, not just per-execution

Conclusions

- ESP-bags algorithm: Extending SP-bags to enable Data-race and Determinism Analysis for async-finish
- Implementation in tool - TaskChecker
- Static Compiler Optimizations reduce overhead
- Average slowdown of 3.05x on a suite of 12 benchmarks

Future Work

- Extend ESP-bags to support more constructs:
  - HJ: futures, phasers
  - X10: futures, clocks, conditional atomics
  - Java: thread fork/join, Java Concurrency Utilities

- Parallelize ESP-bags

- Combine with static determinism verification:

  Automatic Verification of Determinism for Structured Parallel Programs (SAS'10)

Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers

- ***async, finish, asyncMemcpy***

2) Locality control for task and data distribution

- ***hierarchical place tree***

3) Mutual exclusion

- ***ownership-based isolation***

4) Collective, point-to-point, stream synchronization

- ***phasers***

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime and Tools

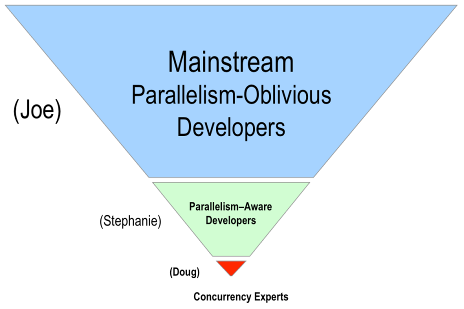
Includes TaskChecker tool!

Two-level programming model

Declarative Coordination Language for Domain Experts, CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Parallelism-aware Developers, Habanero-Java (from X10 v1.5) and Habanero-C



Extreme Scale Platforms

http://habanero.rice.edu