Featherlight Speculative Task Parallelism

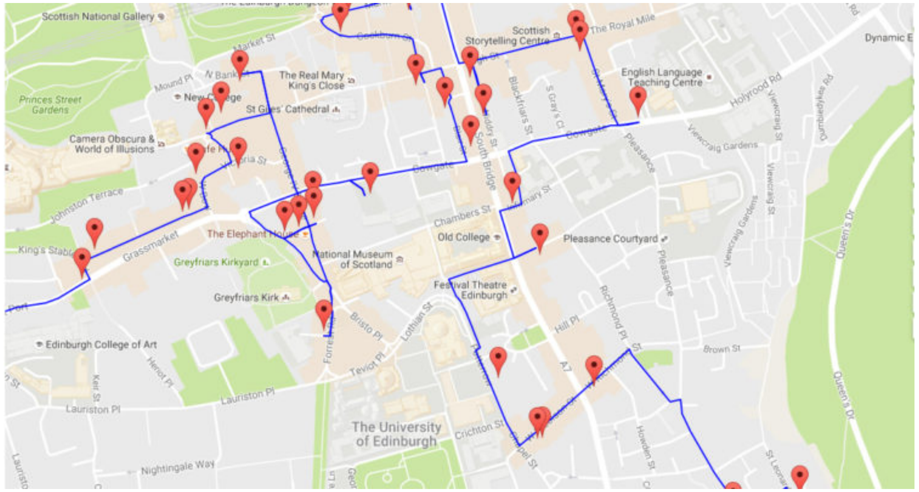
Vivek Kumar

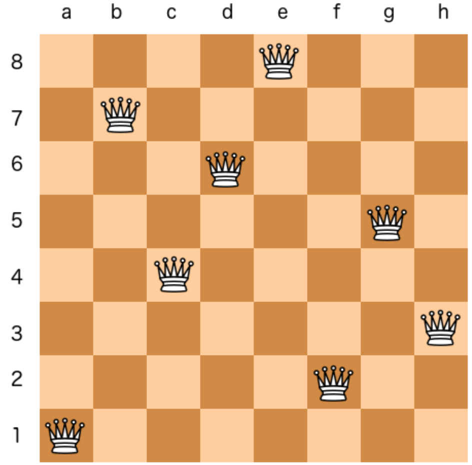
IIIT New Delhi, India

Outline

- Introduction
- Contributions
- Motivating analysis
- Insights and approach
- Implementation
- Experimental Evaluation
- Summary

Goal Based Exploration

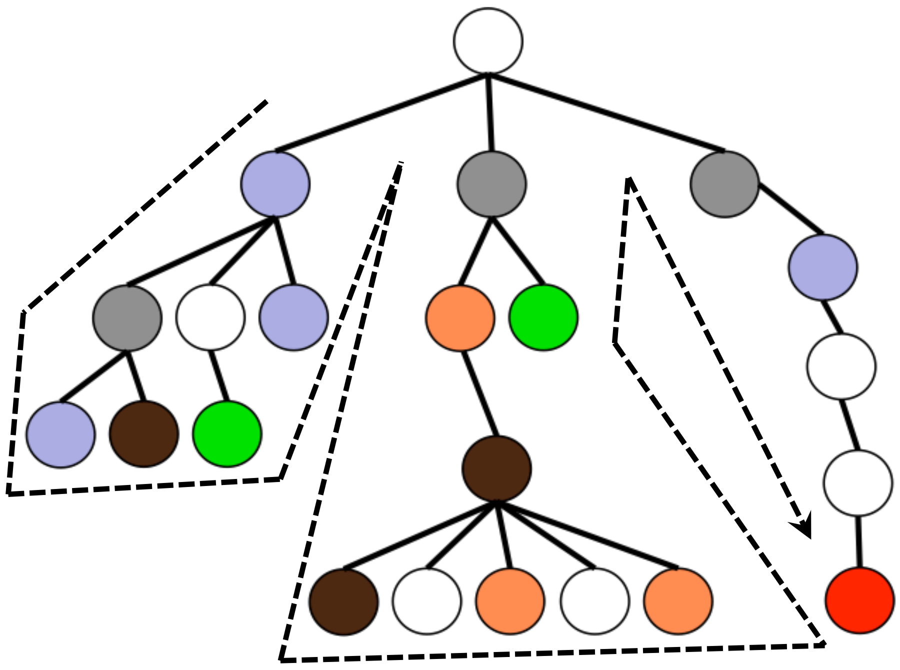




Speculative Parallel Programming

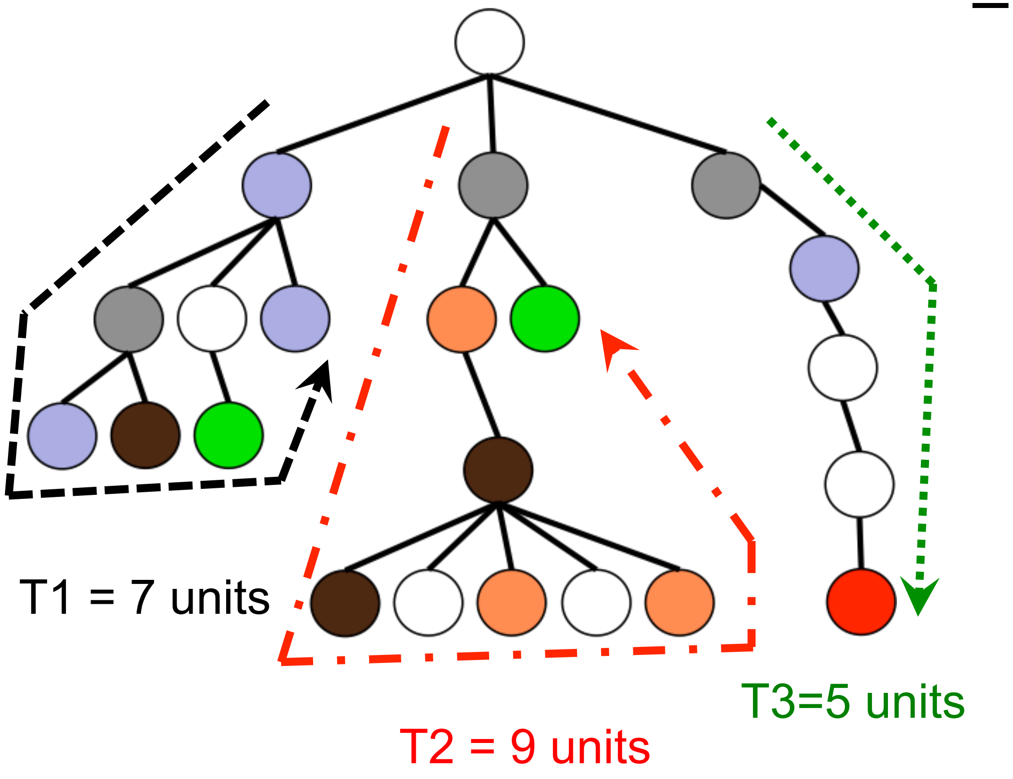
- Parallel programming for goal(s) based exploration
- Not all exploration paths can fetch the expected result(s)
  - Once the goal is found, the search should terminate
  - Highly irregular computation

Goal Based Exploration



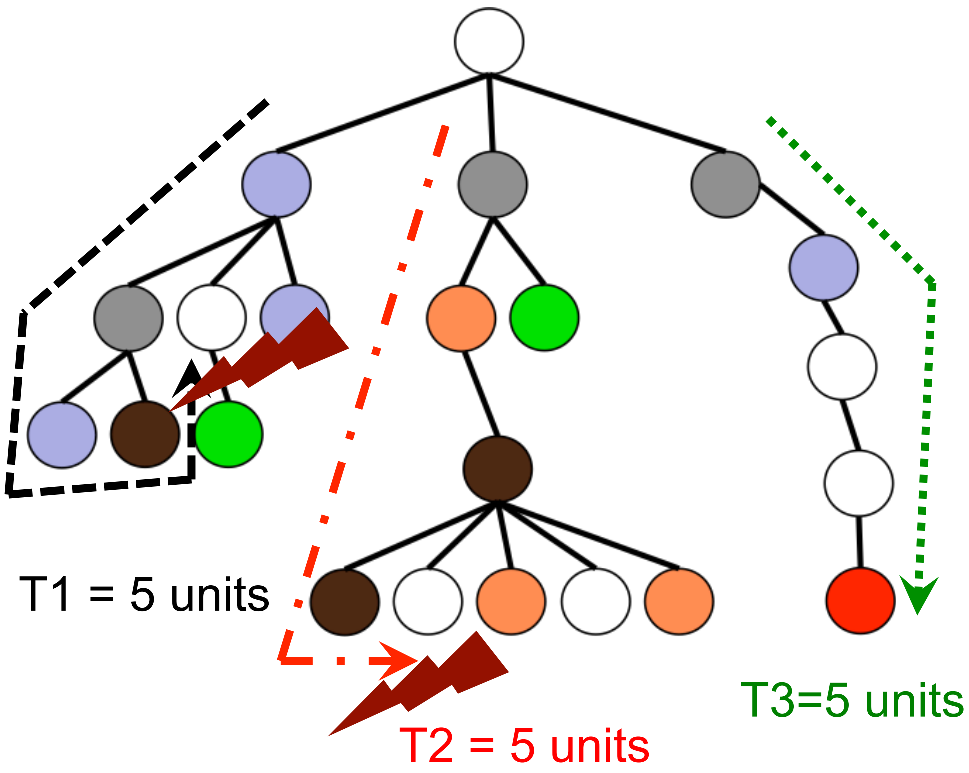
- Unbalanced tree search
  - Search for a unique red node in an unbalanced tree
  - Sequential execution time using DFS
    - 21 units

Goal Based Exploration



- Unbalanced tree search
  - By using 3 threads, one for each outgoing edge from the root?
    - Total execution time (DFS)
      - 9 units (minimum)
      - 4 units of redundant execution

Goal Based Exploration: **Challenges**?



- Software parallelism is difficult to identify and expose
  - Dynamic task parallelism
- How to cancel the redundant execution once a goal is found?
  - Speculative task cancellation

Contributions

✓ Featherlight programming model

For speculative task parallelism that supports serial elision, and doesn’t require task cancellation checks

✓ Lightweight runtime implementation

That leverage mechanisms within modern JVMs

✓ Detailed productivity study

Using a classroom-based study

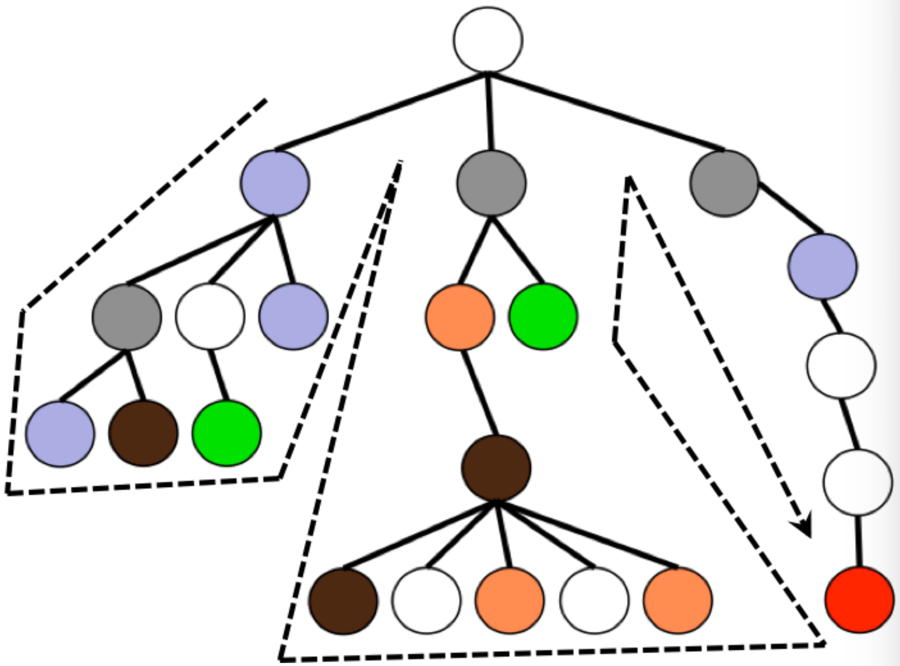
✓ Detailed performance study

Using both micro and real-world benchmarks

Motivating Analysis

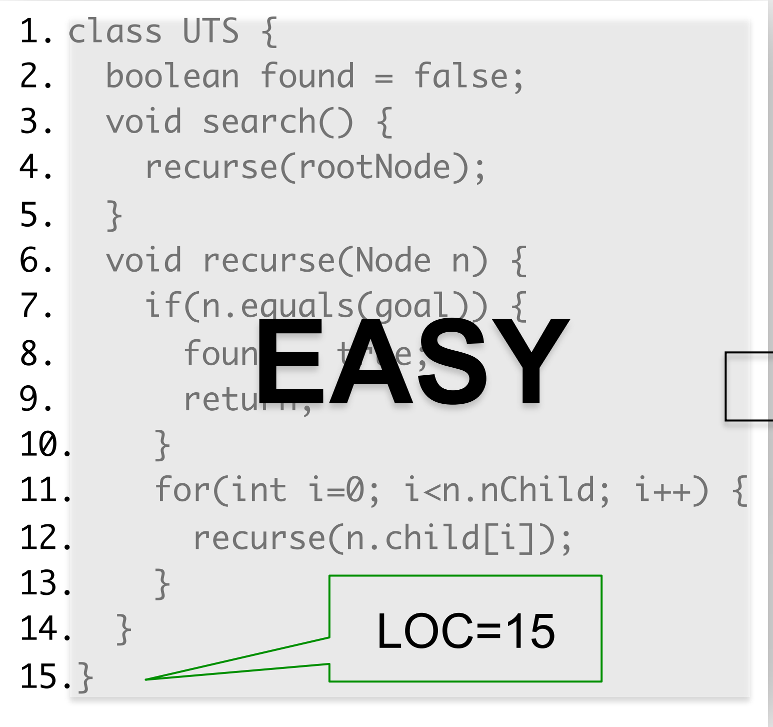
Motivating Analysis

Sequential Unbalanced Tree Search (UTS)



```
class UTS {
  boolean found = false;
  void search() {
    recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
    }
    for(int i=0; i<n.nChild; i++) {
      recurse(n.child[i]);
    }
  }
}
```

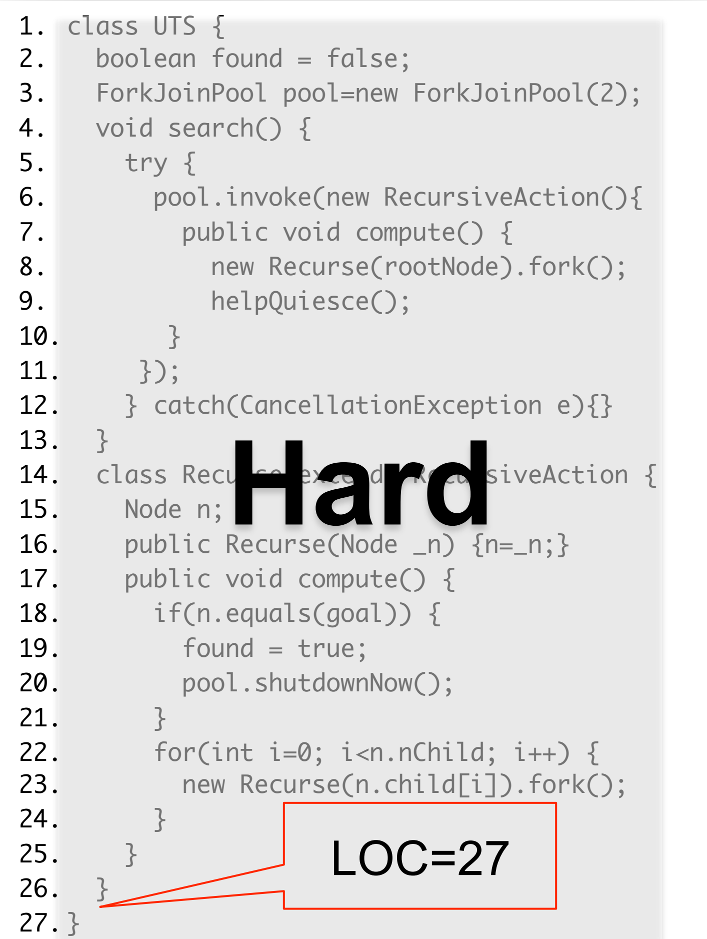
Parallel UTS: Java fork/join



```
class UTS {
  boolean found = false;
  void search() {
    recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
     }
     for(int i=0; i<n.nChild; i++) {
      recurse(n.child[i]);
     }
   }
}
```

EASY

LOC=15



```
class UTS {
  boolean found = false;
  ForkJoinPool pool=new ForkJoinPool(2);
  void search() {
    try {
      pool.invoke(new RecursiveAction(){
        public void compute() {
          new Recurse(rootNode).fork();
          helpQuiesce();
        }
      });
    } catch(CancellationException e){}
  }
  class Recurse extends RecursiveAction {
    Node n;
    public Recurse(Node _n) {n=_n;}
    public void compute() {
      if(n.equals(goal)) {
        found = true;
        pool.shutdownNow();
      }
      for(int i=0; i<n.nChild; i++) {
        new Recurse(n.child[i]).fork();
      }
    }
  }
}
```

Hard

LOC=27

- No serial elision
- Task granularity control required
- Task cancellation checks not required
  - However, applications can't use `try/catch` for `InterruptedException`

Parallel UTS: async-finish (TryCatchWS*)

```
class UTS {
  boolean found = false;
  void search() {
    recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
    }
     for(int i=0; i<n.nChild; i++) {
      recurse(n.child[i]);
     }
  }
}
```

EASY

LOC=15

```
class UTS {
  boolean found = false;
  void search() {
    finish recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
    }
     for(int i=0; i<n.nChild; i++) {
      async recurse(n.child[i]);
     }
  }
}
```

EASY

- Supports serial elision
- Task granularity control not required
- No special support for speculative task parallelism
  - Task cancellation checks required

* *Kumar et al., Work-stealing without the baggage, OOPSLA 2012*

Motivating Analysis

Parallel UTS: async-finish (TryCatchWS*)

```
class UTS {
  boolean found = false;
  void search() {
    recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
    }
     for(int i=0; i<n.nChild; i++) {
       recurse(n.child[i]);
     }
  }
}
```

EASY

```
class UTS {
  AtomicBoolean found = /*allocate*/
  void search() {
    finish recurse(rootNode);
  }
  void recurse(Node n) {
    if(found.get()) return;
    if(n.equals(goal)) {
      found.set(true);
       return;
     }
     for(int i=0; i<n.nChild; i++) {
       if(found.get()) return;
       async recurse(n.child[i]);
     }
  }
}
```

HARD

- Task cancellation checks
  - Inside every method in the call chain
  - Multiple search criteria can complicate the cancellation checks
  - Atomic cancellation tokens
    - May lead to data races if not used properly

** Kumar et al., Work-stealing without the baggage, OOPSLA 2012*

Insights

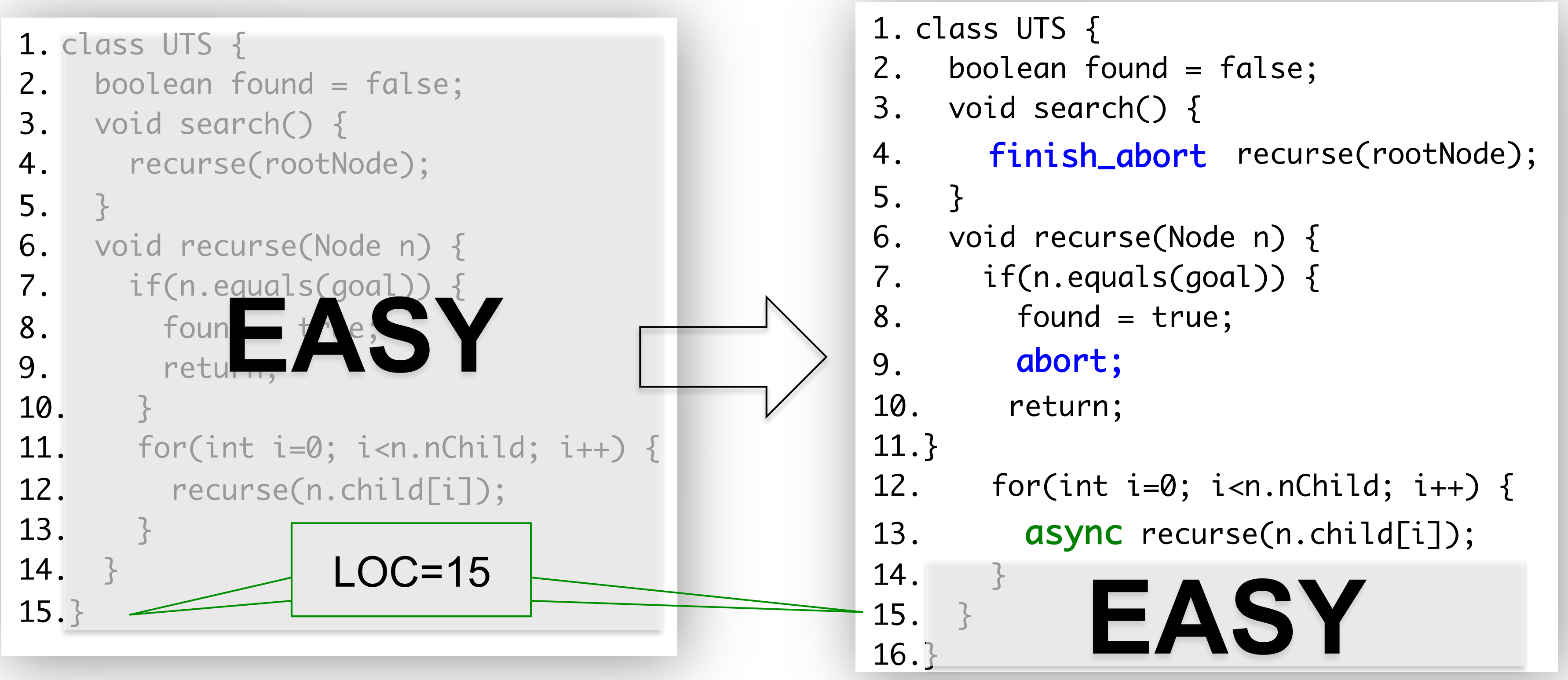
- Cost of tasks cancellation should not incur in common case
- Re-use existing mechanisms inside modern JVMs

Approach

- Cancellation initiation
  - Featherlight programming model
    - ✓`abort` keyword to initiate cancellation
    - ✓`finish_abort` keyword to group tasks searching for same goal
- Handling task cancellation
  - ✓Java exception handling (try–catch blocks)
  - ✓Yieldpoint mechanism to stop running threads
  - ✓Thread stack walk to identify cancelable tasks

Implementation

Featherlight Programming Model



```
class UTS {
  boolean found = false;
  void search() {
    recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      return;
     }
     for(int i=0; i<n.nChild; i++) {
       recurse(n.child[i]);
     }
  }
}
```

```
class UTS {
  boolean found = false;
  void search() {
     finish_abort  recurse(rootNode);
  }
  void recurse(Node n) {
    if(n.equals(goal)) {
      found = true;
      abort;
     return;
}
     for(int i=0; i<n.nChild; i++) {
       async recurse(n.child[i]);
     }
  }
}
```

- Based on TryCatchWS work-stealling runtime
  - Supports serial elision
  - Task granularity control not required
- Task cancellation checks not required
  - `abort` cancels all async tasks **only** within the parent `finish_abort`

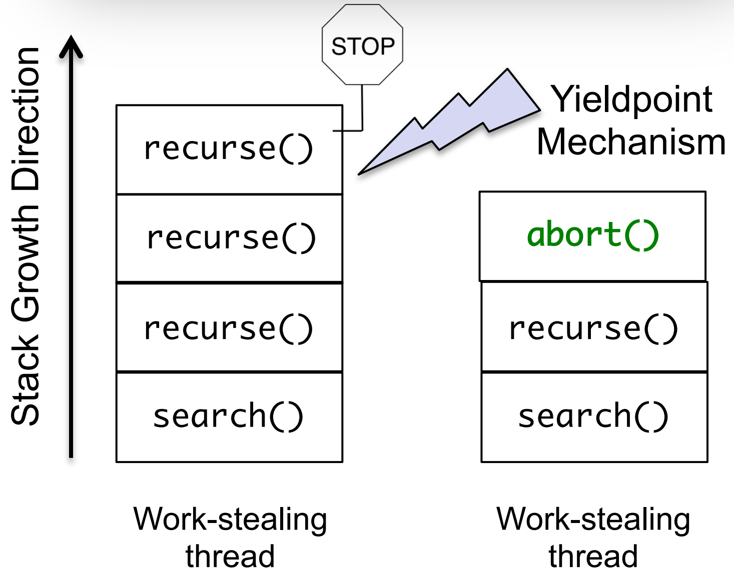
Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint




}
```



Stack Growth Direction

STOP

Yieldpoint Mechanism

recurse()
recurse()
recurse()
search()

Work-stealing thread

abort()
recurse()
search()

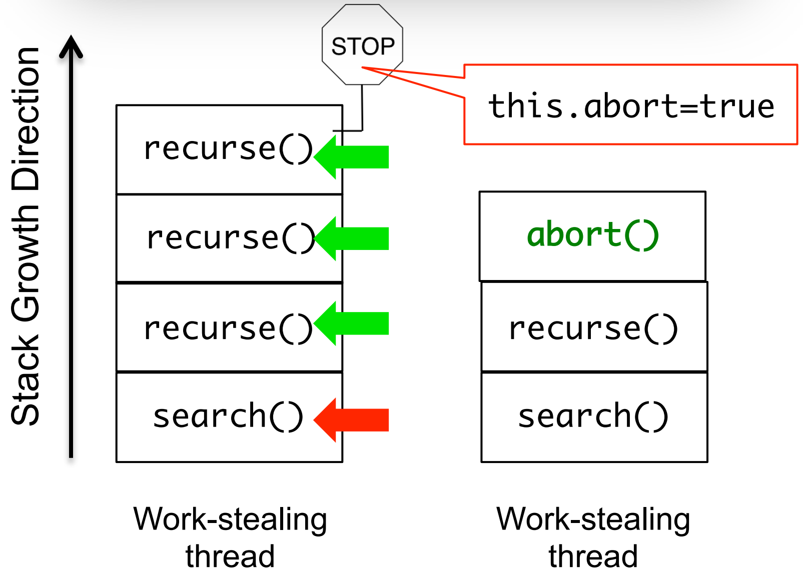
Work-stealing thread

Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true


}
```

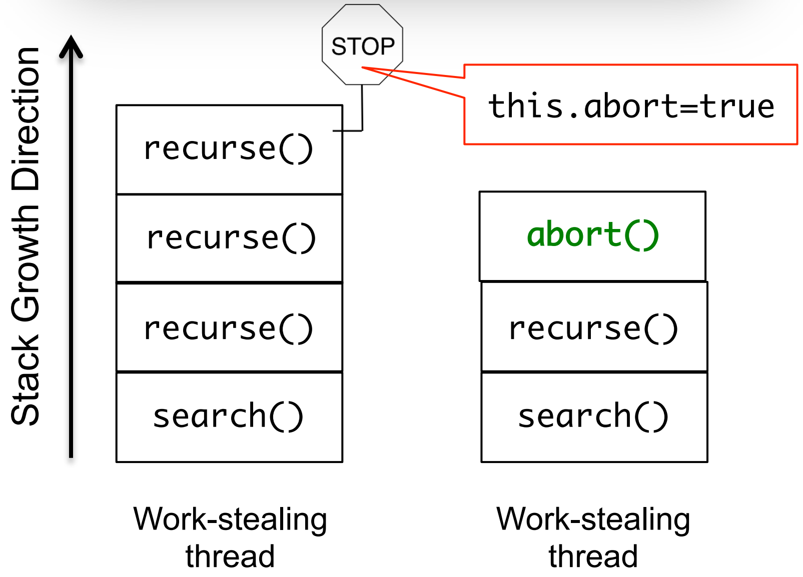


Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true


}
```



Stack Growth Direction

STOP

this.abort=true

recurse()

recurse()

recurse()

search()

Work-stealing thread

abort()

recurse()

search()

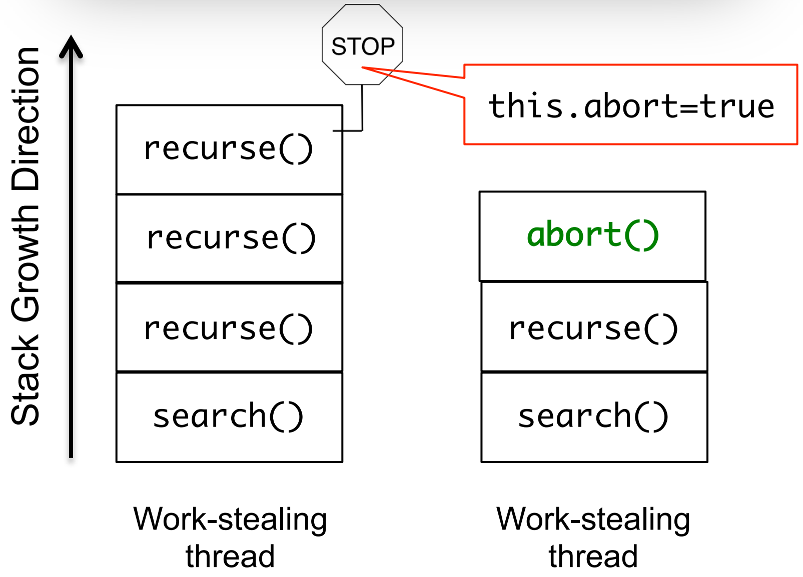
Work-stealing thread

Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true
  // 3. allow “t” to resume from yieldpoint

}
```

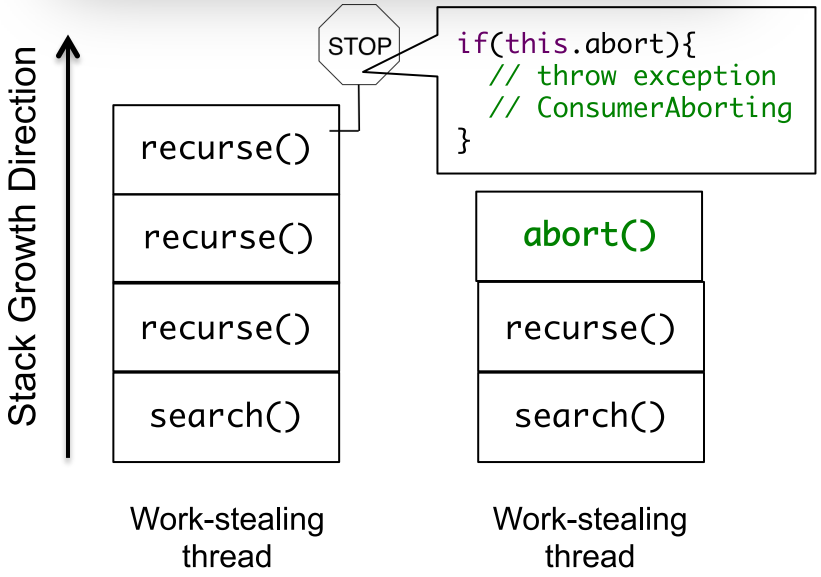


Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true
  // 3. allow “t” to resume from yieldpoint

}
```



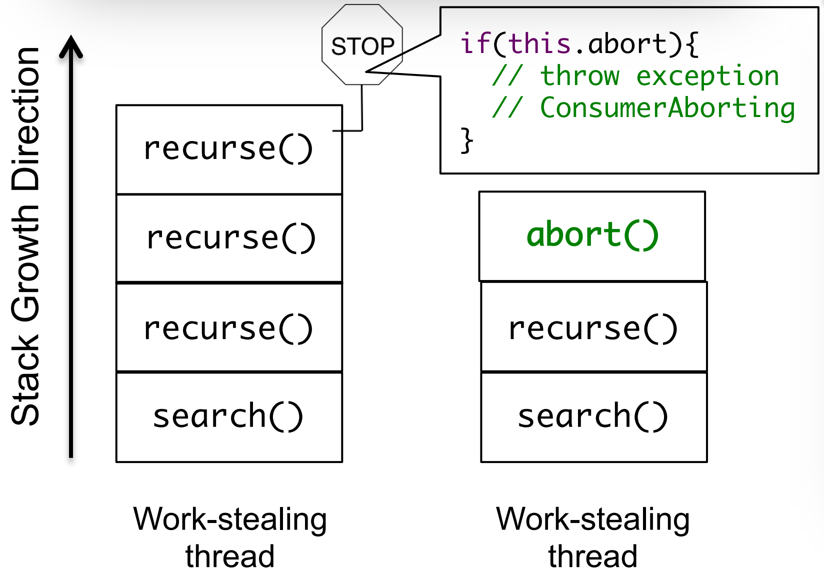
```
if(this.abort){
  // throw exception
  // ConsumerAborting
}
```

Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true
  // 3. allow “t” to resume from yieldpoint

}
```



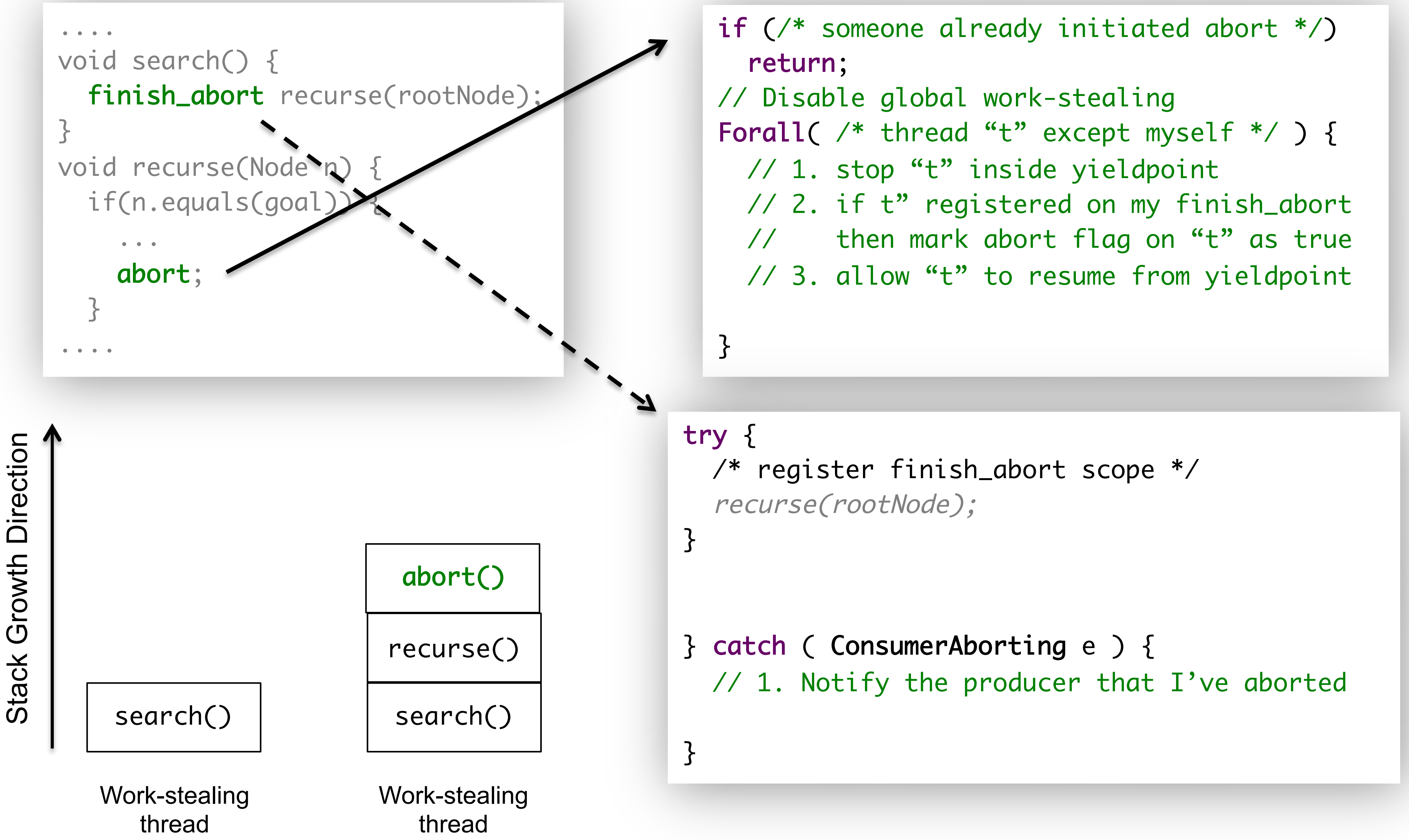
```
if(this.abort){
  // throw exception
  // ConsumerAborting
}
```

```
try {
  /* register finish_abort scope */
  recurse(rootNode);
}


} catch ( ConsumerAborting e ) {


}
```

Featherlight Runtime



```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true
  // 3. allow “t” to resume from yieldpoint

}
```

```
try {
  /* register finish_abort scope */
  recurse(rootNode);
}


} catch ( ConsumerAborting e ) {
  // 1. Notify the producer that I’ve aborted

}
```

Featherlight Runtime

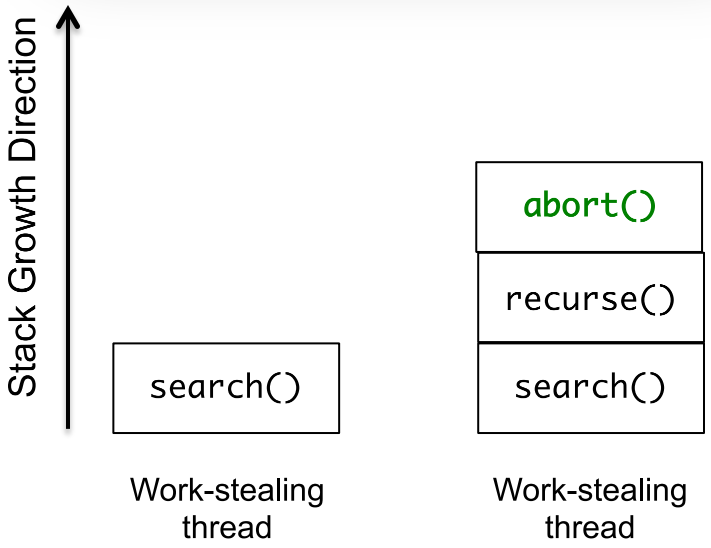
```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread "t" except myself */ ) {
  // 1. stop "t" inside yieldpoint
  // 2. if t" registered on my finish_abort
  //    then mark abort flag on "t" as true
  // 3. allow "t" to resume from yieldpoint
  // 4. throw exception "ProducerAborting"
}
```

```
try {
  /* register finish_abort scope */
  recurse(rootNode);
} catch ( ProducerAborting e ) {


} catch ( ConsumerAborting e ) {
  // 1. Notify the producer that I've aborted

}
```



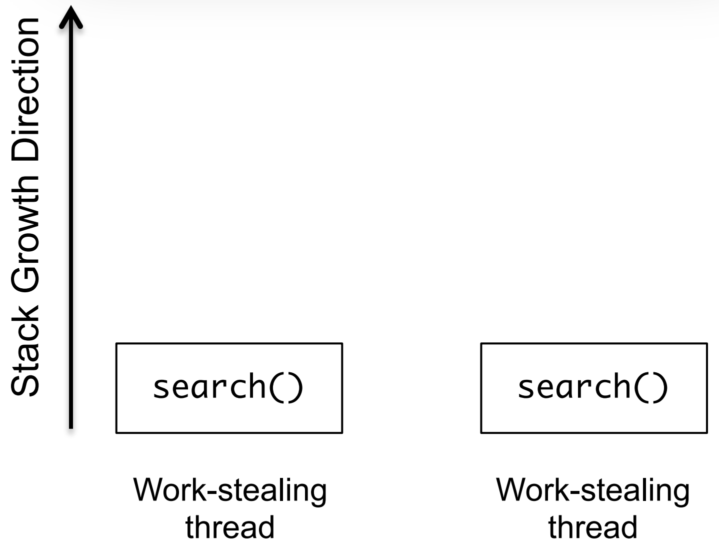
Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread “t” except myself */ ) {
  // 1. stop “t” inside yieldpoint
  // 2. if t” registered on my finish_abort
  //    then mark abort flag on “t” as true
  // 3. allow “t” to resume from yieldpoint
  // 4. throw exception “ProducerAborting”
}
```

```
try {
  /* register finish_abort scope */
  recurse(rootNode);
} catch ( ProducerAborting e ) {
  // 1. Wait until all relevant workers aborted
  // 2. Enable global work-stealing
} catch ( ConsumerAborting e ) {
  // 1. Notify the producer that I’ve aborted

}
```

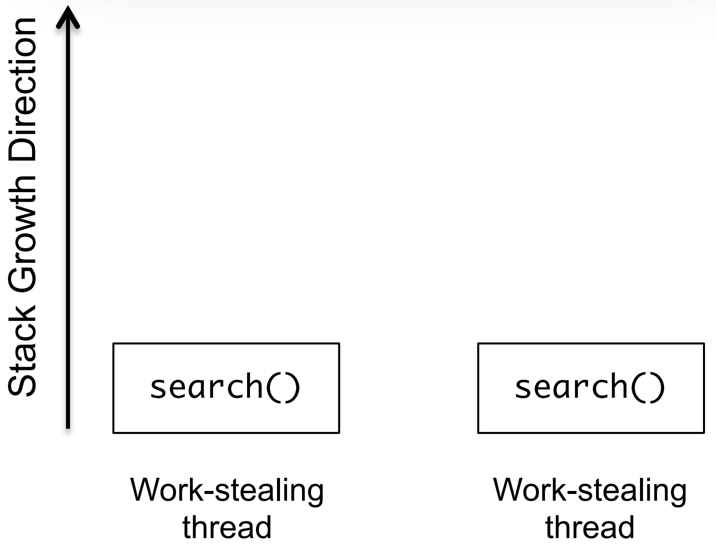


Featherlight Runtime

```
....
void search() {
  finish_abort recurse(rootNode);
}
void recurse(Node n) {
  if(n.equals(goal)) {
    ...
    abort;
  }
....
```

```
if (/* someone already initiated abort */)
  return;
// Disable global work-stealing
Forall( /* thread "t" except myself */ ) {
  // 1. stop "t" inside yieldpoint
  // 2. if t" registered on my finish_abort
  //    then mark abort flag on "t" as true
  // 3. allow "t" to resume from yieldpoint
  // 4. throw exception "ProducerAborting"
}
```

```
try {
  /* register finish_abort scope */
  recurse(rootNode);
} catch ( ProducerAborting e ) {
  // 1. Wait until all relevant workers aborted
  // 2. Enable global work-stealing
} catch ( ConsumerAborting e ) {
  // 1. Notify the producer that I've aborted
  // 2. Start stealing task from others
}
```



Experimental Evaluation

Methodology

- Benchmarks
  - Goal based exploration
  - Micro kernels
    - UTS
    - LinearSearch
    - NQueens
    - ShortLongPath
    - Sudoku
    - TravelingSalesman
  - Real-world
    - Dacapo lusearch-fix

- Hardware Platform
  - 2 Intel Xeon E5-2650
    - 10 cores each

- Software Platform
  - Jikes RVM

Runtime+Benchmarks: *https://github.com/hipec/ featherlight/archive/4075770.tar.gz*

Productivity Analysis (1/2)

- Extra LoC compared to Sequential version

Benchmark	Common Code	Sequential	Featherlight	ManualAbort	Java ForkJoin
UTS	545	39	0	6	19
LinearSearch	88	44	0	2	31
NQueens	75	48	0	5	20
ShortLongPath	558	54	0	6	22
Sudoku	469	48	0	6	18
Traveling Salesman	158	55	0	6	29
Dacapo lusearch-fix	>126K	222	0	20	19

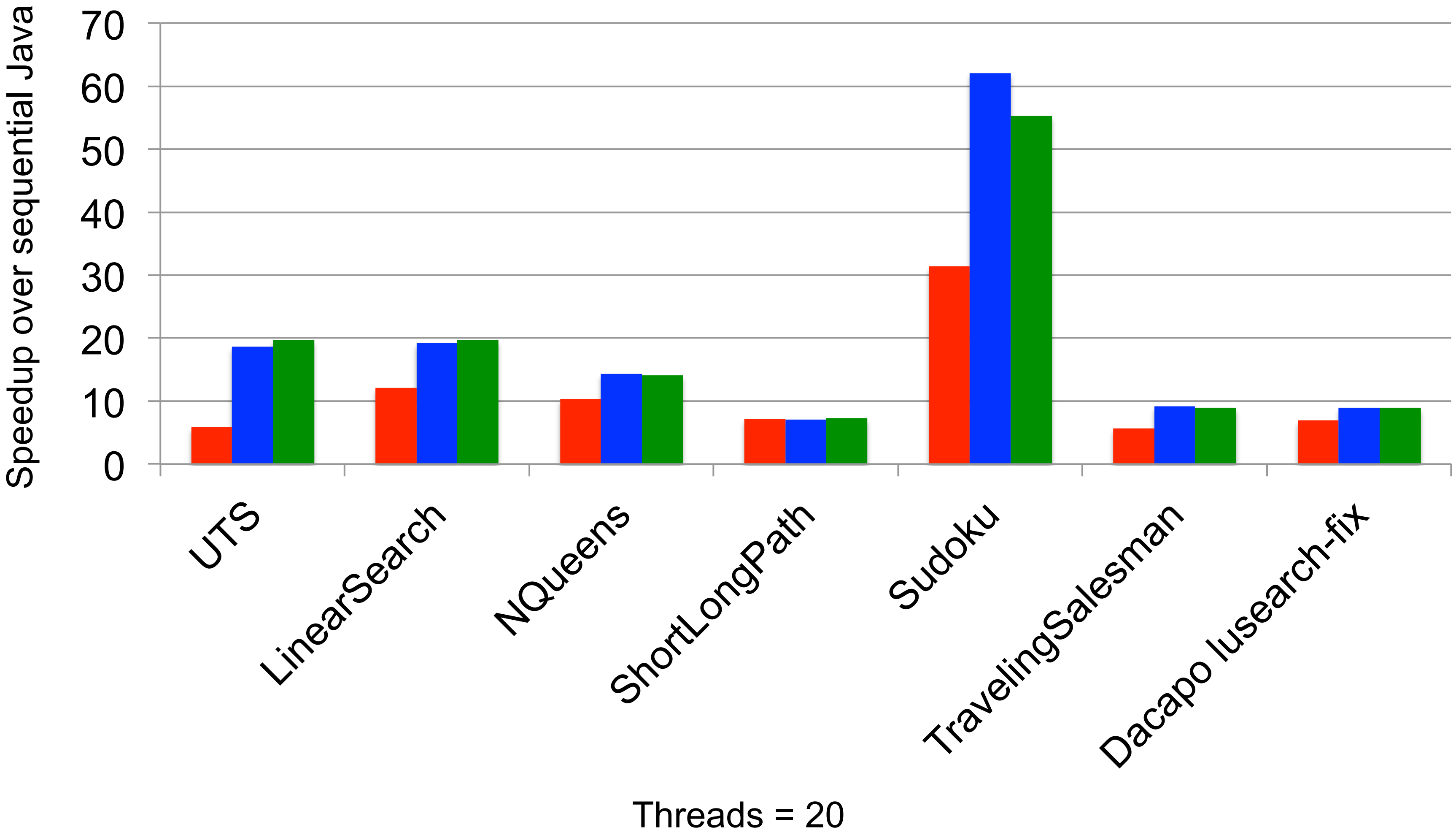
Productivity Analysis (2/2)

- Time (minutes) spent by students in classroom for implementing the parallel versions

Benchmark	Subjects	Mean		Subjects	Mean
UTS	9	**8.6**		9	52.2
LinearSearch					
NQueens	7	**13.6**		8	61
ShortLongPath	6	**11.7**		8	43.4
Sudoku	6	**6**		8	58.8
Traveling Salesman	7	**10.4**		8	53.1
Dacapo lusearch-fix					
	Featherlight			**Java Fork/Join**	

LinearSearch was not included as both its parallel implementations were provided as examples.
Dacapo was not included due to its cumbersome setup.

Performance Analysis



Summary and Conclusion

- Speculative task parallelism
  - Task cancellation checks reduce productivity
- Featherlight
  - Automatic cancellation of speculative tasks
    - Improves productivity without degrading performance
  - `finish_abort`
    - Synchronization and grouping of cancelable tasks
    - Uses try/catch blocks
  - `abort`
    - Initiates cancellation
    - Reuses existing mechanism inside modern JVMs
      - Yieldpoint mechanism to stop the threads
      - Stack walk to identify cancelable tasks