



Hint-based Static Analysis for Automatic Program Parallelization



Steve Simon Joseph Fernandez
CSCI 5535
University of Colorado, Boulder



Background

Year	Hardware	Software (mostly)
1998	Single Core	Single worker thread, 100% CPU utilization when needed.
2008	2x Quad Core (16 hardware threads)	???



Why parallelize?

- Because applications demand it:
 - Simulation
 - Climate prediction
 - DNA sequencing
 - 3D modeling
- Because hardware supports it:
 - Intel Nehalem “Beckton” – 8 processor cores, 16 hardware threads
 - nVidia GTX 295 – 480 processor cores



What to parallelize?

- Loops, of course!
 - Imperative programming – majority of “work” done in loops
- Other code constructions can be parallelized too
 - But focus is on loops for now.



Why *automated* parallelization?

- Because an automated system
 - Can be guaranteed error-free
 - Can apply optimizations specific to target hardware
 - Can free the programmer from minute parallelization details
- Because of “Endless Possibilities”:
 - Could parallelize all “legacy” code, resulting in tremendous performance gains without any additional human cost



So, what's the problem?

- Static analysis is *hard*
 - Undecidable in many cases
 - Memory and processor intensive
 - Trade-off depending on the abstraction used



What makes Static Analysis “hard”?

- Choosing the “right” abstraction
- Not “knowing” on “what” to focus the analysis on

Static analysis for a simple *for* loop involves:

- Alias analysis
- Data dependence analysis
 - Flow dependence
 - Antidependence
 - Output dependence
 - Input dependence

For a nested loop with n layers, data dependence analysis would require solving equations in n -dimensional spaces.



So, can we do something about it?

- *Who's "we"?* – The programmer, the author of the source code.
- *Why the programmer?* – Because (s)he “knows” the source code better than anyone else.
- *What can the programmer do?* – Guide the analysis to focus on the aspects of the source code we're interested in.
- *Why do we want to do it?* – To get the most out of the analysis, w.r.t. precision and correctness, taking into consideration time, processing power and memory limitations.
- *How can it be done?*



One possible solution

Annotate the source with instructions and hints to the analyzer, which are used during the analysis.

During the analysis, the analyzer would also ask questions at point where it has insufficient data.



“Hard” areas in static analysis

- Alias analysis
 - Hard to determine whether variables / elements in a data structure are aliases
 - Aliasing influences optimization decisions in a big way
- Range analysis
 - Hard to infer the possible values of variables that are of interest within a loop, e.g. the range of the loop counter
 - Often, the programmer may have a very good idea of the lower and/or upper bounds of variables used inside a loop
- Invariant analysis
 - Hard to infer the invariant(s) with the “exact” amount of information needed



Suggested annotations (thus far)

Language under consideration: Java

- @Parallelize
- @Independent
- @Range
- @Invariant
- @Alias



@Parallelize

- Can be used to indicate loops to be parallelized
- Analyzer would use information available at that point (including those from other annotations) to determine whether that loop is parallelizable or not.
- All other annotations would follow @Parallelize

Example: Instructing to parallelize only the innermost loop in a nested for-loop

```
for(i=0; i<loop1; i++)
  for(j=0; j<loop2; j++)
    @Parallelize
    for(k=0; k<loop3; k++)
      .....
```



@Independent(p1, p2...)

- Used to indicate operations that are inherently parallel, but might be difficult to prove using analysis
- Can also be used to indicate (using a pattern) which loop iterations are independent of each other

Example: Indicating that the innermost loop in a nested series of loops is inherently parallel

```
for(i=0; i<loop1; i++)
  @Parallelize
  for(j=0; j<loop2; j++)
    @Independent
    for(k=0; k<loop3; k++)
      .....
```



@Range(var, [lbExpression], [ubExpression])

- Indicates the range of possible values the variable var can take inside the loop
- Can suggest just the lower bound, upper bound or both
- The bounds can be expressed in terms of other variables inside the loop

Example:

```
for(i=0; i<loop1; i++)  
  for(j=0; j<loop2; j++)  
    @Parallelize  
      @Range(k, 0, 10000)  
        for(k=start; k<end; k++)  
          .....
```



@Invariant(expression)

- Can be used to provide loop invariants
- Can be used at the loop level or at the statement level
- (More research needed on this one)



`@Alias(varExpression1,
varExpression2)`

- Indicates that `varExpression1` and `varExpression2` are aliases for the same data/memory location
- `varExpression` may contain inductive/enumerative definitions
- (More research needed on this one)

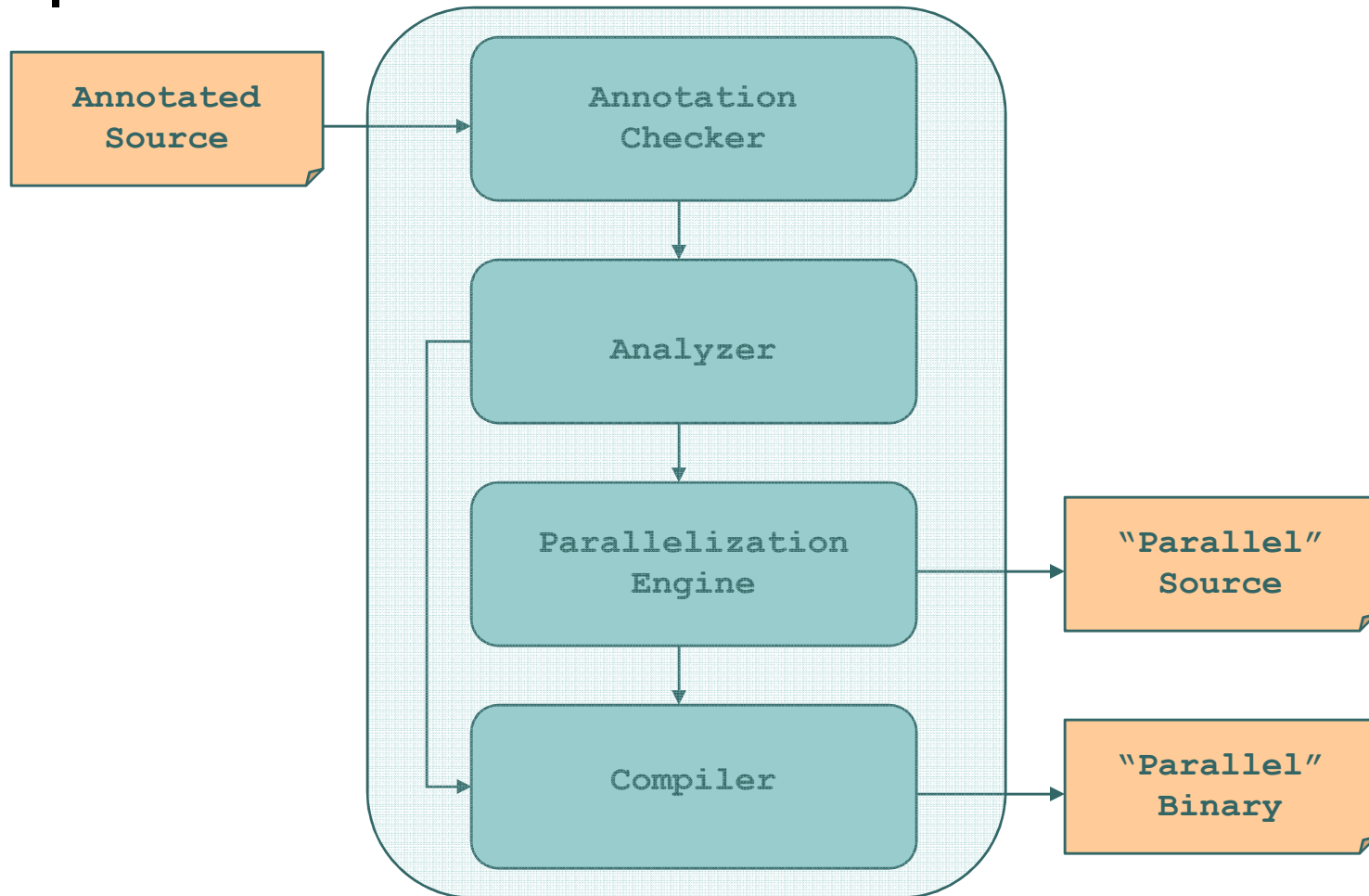


How does this info help in parallelization?

- These hints provide direct/indirect answers to most of the questions that arise during program optimizations and transformations
- These hints, along with more information from static analysis, could be used to:
 - Generate parallelized source code
 - Generate binaries optimized for a particular platform



Modus Operandi





Drawbacks

- “Correctness” of the annotations
 - Need syntax checking
 - Need semantic checking
- Analyzer bases all its decisions on the info in the annotations; errors that modify program behavior could be costly
- Annotation specification requires an in-depth knowledge about the “meaning” of the source
- Knowledge about analysis concepts also needed



Summary

- In order to make legacy code take advantage of multiple hardware threads, we would like to parallelize it by some automated process
- Static analysis on source code is broad, time consuming, resource intensive and often undecidable
- The programmer knows the areas of code that would benefit from parallelization, so (s)he can “guide” the analysis to focus in the right direction
- One possible way of doing so is to annotate the code with instructions/hints to the analyzer
- Annotations indicate properties of data within the structure to be parallelized; not concerned about the properties at other locations
- Based on a survey of papers, some possible hints outlined here



Future work

- Search for more properties that would be useful in parallelization but hard to infer statically, and provide annotations for the same
- Adapt an existing static analyzer to consume these annotations and examine what other useful inferences it produces
- Investigate the types of guarantees provided by such an analysis
- Investigate how this approach can be combined with other runtime-based approaches to further optimize parallelism



Questions?

