Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis

Rishi Surendran
Rice University
Houston, TX, USA
rishi@rice.edu

Vivek Sarkar
Rice University
Houston, TX, USA
vsarkar@rice.edu

Abstract

We introduce a novel approach for using futures to automatically parallelize the execution of pure method calls. Our approach is built on three new techniques to address the challenge of automatic parallelization via future synthesis: candidate future synthesis, parallelism benefit analysis, and threshold expression synthesis. During *candidate future synthesis*, our system annotates pure method calls as async expressions and synthesizes a parallel program with future objects and their type declarations. Next, the system performs a *parallel benefit analysis* to determine which async expressions may need to be executed sequentially due to overhead reasons, based on execution profile information collected from multiple test inputs. Finally, *threshold expression synthesis* uses the output from parallelism benefit analysis to synthesize predicate expressions that can be used to determine at runtime if a specific pure method call should be executed sequentially or in parallel.

We have implemented our approach, and the results obtained from an experimental evaluation of the complete system on a range of sequential Java benchmarks are very encouraging. Our evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×) relative to the sequential programs when using 8 processor cores, with zero programmer effort beyond providing the sequential program and test cases for parallelism benefit analysis.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis, Program Transformation

Keywords Futures, Automatic Parallelization

1. Introduction

Parallelizing programs to effectively utilize multicore architectures is a major challenge facing application developers and domain experts. In this paper, we introduce a novel approach for automatically parallelizing pure method calls using futures as the primary parallel construct. A method is pure [18, 32] if it (or any method that it calls) does not mutate any object in the program state that exists before the method is invoked. However, a pure method is permitted to mutate objects that are allocated during its execution and return a newly constructed object as the result. Further, a pure method is allowed to read global state that may be later mutated by the method's caller.

Automatic parallelization is a very challenging problem in general. As an example, the current state of the art in automatic parallelization of loops with array accesses (including program dependence graphs [12] and polyhedral frameworks [11]) was developed over four decades with many restrictions along the way with respect to array subscript expressions and procedure calls in loops. The seminal papers in this field (e.g. [21]) included results for simple loop nests, and it took many years for the original ideas to be refined and applied to real-world programs. This paper is the first to address the problem of automatic parallelization by generating futures, which is different from past work on automatic loop/statement-level parallelization using control and data dependences since future references can be copied without waiting for the result to be computed.

A future [15] (or promise [22]) refers to an object that acts as a proxy for a value, because the computation of the value may still be in progress as a parallel task. In the notation used in this paper, the statement, "`future<T> f = async<T> Expr;`" creates a new child task to evaluate the expression `Expr` asynchronously, where `T` is the type of the expression `Expr`. It also assigns to f a reference to a handle (future object) for the return value from `Expr`. The operation

```
1  class TreeNode {
2    private TreeNode left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        TreeNode l = bottomUpTree(2*item-1,
             depth-1);
7        TreeNode r = bottomUpTree(2*item, depth-1);
8        return new TreeNode(l, r, item);
9      } else {
10       return new TreeNode(item);
11     }
12   }
13   ...
14   int itemCheck() {
15     ...
16     return item + left.itemCheck() -
           right.itemCheck();
17   }
18 }
```

Figure 1: Sequential Binary Tree program [23] from Computer Language Benchmarks Game

```
1  class TreeNode {
2    private mayfuture<TreeNode> left, right;
3    static int THRESHOLD = 12;
4    ...
5    TreeNode bottomUpTree(int item, int depth){
6      if (depth > 0) {
7        mayfuture<TreeNode> l, r;
8        if (depth-1 >= THRESHOLD) {
9          l = async<TreeNode> {
10           return bottomUpTree(2*item-1, depth-1);
11         }
12       } else
13         l = bottomUpTree(2*item-1, depth-1);
14       if (depth-1 >= THRESHOLD) {
15         r = async<TreeNode> {
16           return bottomUpTree(2*item, depth-1);
17         }
18       } else
19         r = bottomUpTree(2*item, depth-1);
20       return new TreeNode(l, r, item);
21     } else {
22       return new TreeNode(item);
23     }
24   }
25   ...
26   int itemCheck() {
27     ...
28     return item + left.get().itemCheck() -
           right.get().itemCheck();
29   }
30 }
```

Figure 2: Binary Tree program from Figure 1 after parallelization using the approach presented in this paper. Our implementation does the transformations on Java bytecode. The equivalent source code is shown here.

`f.get()` can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available. There are a number of runtime approaches (e.g., [19]) that can be employed to reduce the overhead of the blocking operations related to futures.

Futures are traditionally used for enabling functional style parallelism, and therefore, are a natural fit for parallelizing the execution of pure method calls. They also have the advantage that references to future objects can be copied without waiting for the future tasks to have completed, thereby exposing more parallelism than in imperative-style task parallel constructs. Finally, the synchronization patterns that can be expressed by structured fork-join models (such as OpenMP's task parallelism [26], Cilk's spawn-sync [2] parallelism) are inherently limited to series-parallel computation graphs, while futures can be used to generate any arbitrary computation graph.

As an example, consider the program from the Computer Language Benchmarks Game [23] in Figure 1, which constructs a binary tree using method `bottomUpTree` and then performs a traversal of the tree using method `itemCheck`. The program after parallelization using the approach presented in this paper (using a test input that constructs a tree with height = 14) is shown in Figure 2. The parallelization algorithm made the following changes to the program: 1) The construction of the tree is performed as future tasks, if the current depth[1] is greater than or equal to a certain threshold; 2) The types of the fields `left` and `right` and variables `l` and `r` are changed from `TreeNode` to `mayfuture<TreeNode>`, where `mayfuture<T>` may refer to a `future<T>` object or an object of type `T`; 3) A new constructor is added to the `TreeNode` class which accepts `mayfuture<TreeNode>` as its first and second arguments; and, 4) `get()` calls are inserted before the result of a future/mayfuture object is used. Another important aspect of our approach is that even though both `bottomUpTree` and `itemCheck` are pure methods, our approach only parallelizes the execution of the `bottomUpTree` method since the work performed by `itemCheck` is not profitable for parallelization. Compared to using imperative-style task parallel constructs, this program has more parallelism because it performs a `get()` only when the result of the future task is used in line 28 of Figure 2. If the same program is parallelized using imperative-style constructs such as `spawn-sync` or `async-finish`, the parallelized program will require synchronization to ensure that the tasks created in line 9 and line 15 complete before the constructor invocation in line 20 of Figure 2.

In summary, the main contributions of this paper are as follows:

- A static analysis algorithm for *future synthesis* that can be used to synthesize a parallel program with future objects, their type declarations, and async expressions. Our approach synthesizes object clones when needed, and generates more precise type information for future objects, compared to future synthesis algorithms reported in past work for manual parallelization.
- A *parallelism benefit analysis* algorithm, which determines the profitability of executing a method call as a

[1] This benchmark uses a parameter named depth for what might usually be considered to be the height of the node. For example, the depth parameter is zero for all leaf nodes.

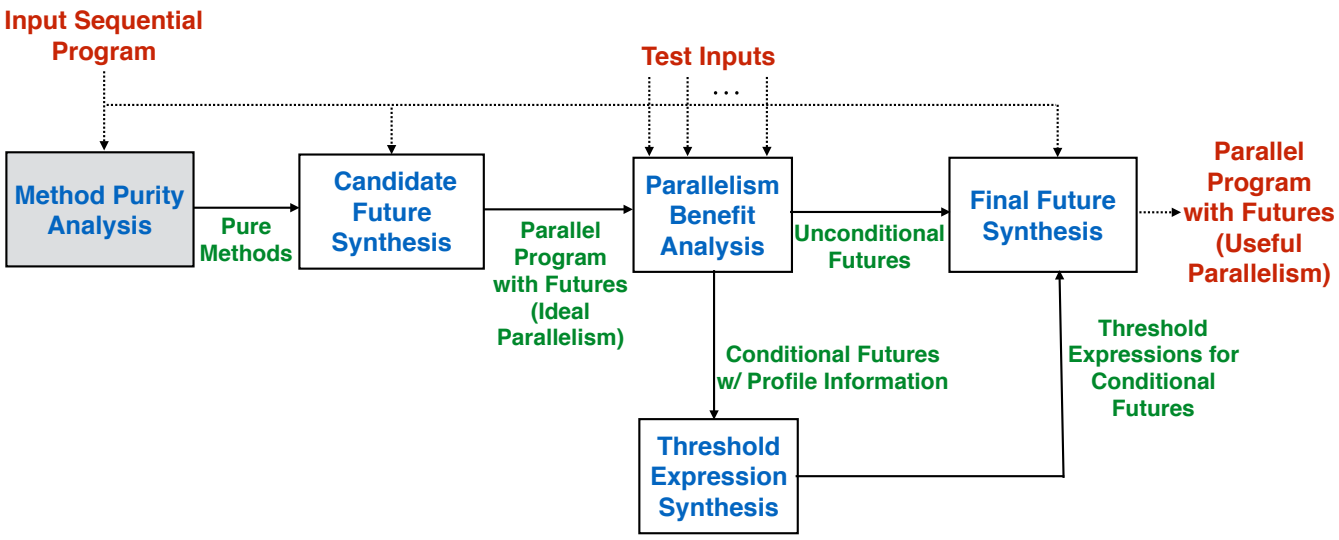

Figure 3: High-level view of our approach. The dotted lines represent user inputs and outputs. The grey box (Method Purity Analysis) represents past work leveraged by our approach, whereas the other boxes represent new contributions.

future task. The analysis is based on execution profile information collected from multiple test inputs.

- An algorithm to synthesize *threshold conditional expressions*, which determine dynamically whether a specific method call should be executed sequentially or in parallel.
- These algorithms have been implemented as analyses and transformations to generate parallel Habanero Java (HJ) [6] code from sequential Java code, and evaluated on a range of benchmark programs. When using 8 processor cores, the evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×).

The rest of the paper is organized as follows. Section 2 presents an overview of our approach. Sections 3-6 describe the technical details of our solution. Section 7 contains our experiment results. Section 8 discusses related work, and Section 9 summarizes our conclusions.

2. Overview of Our Approach

A high-level view of our approach is given in Figure 3. The five main steps in automatic parallelization of eligible method calls are as follows:

1. **Method Purity Analysis**: The first step in our approach is the identification of pure methods. Our implementation uses past work (ReImInfer [18]) on automatic purity analysis to identify pure methods in Java programs, but can also be applied to programs in which methods are annotated as pure by the programmer.
2. **Candidate Future Synthesis (CFS)**: Our tool annotates calls to a subset of the pure methods identified by ReImInfer as async expressions. (The subset focuses on methods containing iterative/recursive subcomputations, so as not to overwhelm later instrumentation phases with trivial and unprofitable candidates for execution as future tasks.) Next, we generate a parallel program by synthesizing futures from the async expressions. The synthesis algorithm is presented in Section 3 and involves two steps: 1) inter-procedural future analysis, which determines the locations in the input program where a future object may be accessed, as well as the types of the future objects and 2) future transformations, which changes the types of future objects and inserts future get operations.
3. **Parallelism Benefit Analysis (PBA)**: Once we have a parallel program with futures, we construct Weighted Computation Graphs (WCGs) for the program for each of the given test inputs. (The choice of test inputs only impacts the performance, not the correctness, of parallelization.) The weights of the nodes in the WCG represent the work done by each of the steps and the overheads of task creation, task termination and synchronization operations. The weighted computation graphs are then analyzed to identify tasks that provide benefit from parallelization. Based on the analysis results, each method call site is classified as *serial*, *parallel* or *conditional parallel*. The parallelism benefit analysis algorithm is presented in Section 4.
4. **Threshold Expression Synthesis (TES)**: For call sites that are identified as conditional parallel by PBA, this step synthesizes an expression that enables conditional parallel execution of method invocations. The threshold expression identifies a subset of method invocations at the call site, for which the work done by the method is greater than a certain threshold, which we refer to as

```
1  class TreeNode {
2    private TreeNode left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        TreeNode l = async bottomUpTree(2*item-1,
             depth-1);
7        TreeNode r = async bottomUpTree(2*item,
             depth-1);
8        return new TreeNode(l, r, item);
9      } else {
10       return new TreeNode(item);
11     }
12   }
13   ...
14   int itemCheck() {
15     ...
16     return item + async left.itemCheck() - async
           right.itemCheck();
17   }
18 }
```

Figure 4: Binary Tree program from Figure 1 after pure function analysis and `async` expression annotation

sequential threshold. The threshold expression synthesis algorithm is presented in Section 5.

5. **Final Future Synthesis**: In the last step, we generate parallel code from the input sequential program based on the analysis done by the previous steps. This involves cloning inputs when needed, annotation of parallel call sites as async expressions, and conditional annotation of conditional parallel call sites as async expressions. The final future synthesis step is described in Section 6.

3. Candidate Future Synthesis

In this section, we present our approach for synthesizing futures in a program annotated with `async` expressions. The `async` expressions that serve as the source for synthesis are inserted based on the output of method purity analysis. The program from Figure 1 after async expression annotation using method purity analysis is shown in Figure 4. The two functions `bottomUpTree` and `itemCheck` do not cause any side-effects, and their calls are therefore marked as `async` expressions. The async expression annotated program is then passed as input for future synthesis.

The synthesis process involves the following steps:

1. Replacing `async` expressions by typed `future` expressions.
2. Identifying inputs that need to be cloned in the final future synthesis step.
3. Analyzing the whole program and modifying the types of variables and fields that can refer to a future object. Their type is changed from `T` to `future<T>`, if the variable or field *must* refer to a future at all program points where the variable/field is accessed. If the variable or field *may* refer to a future, the type is changed to `mayfuture<T>`. This is in contrast to past work [27] in which all future variables/fields were declared with an `Object` type in both cases, and cast operations were inserted whenever they needed to be accessed as future objects (Section 3.1).
4. Identify methods that perform non-constant amount of work and are candidates for asynchronous execution (Section 3.2).
5. Insertion of calls to `get()` before the result of a future task is used, along with `instanceof` checks when needed for `mayfuture` objects.

Statement	Data flow function
`t := async e;`	$\lambda Y.(Y \cup \{\mathtt{t}\})$
`x := new` τ`;`	$\lambda Y.(Y - \{\mathtt{x}\})$
`x.f := t;`	$\lambda Y.(\text{if } \mathtt{t} \in Y \text{ then } Y \cup \{\mathtt{f}\} \text{ else } Y)$
`t := x.f;`	$\lambda Y.(\text{if } \mathtt{f} \in Y \text{ then } Y \cup \{\mathtt{t}\} \text{ else } Y)$
`x := t;`	$\lambda Y.(\text{if } \mathtt{t} \in Y \text{ then } Y \cup \{\mathtt{x}\} \text{ else } Y)$

Figure 5: Examples of normal flow function for computing $\mathcal{M}$ (may-be-future)

Section 3.1 presents an inter-procedural data flow analysis that identifies the locations in the program where a future object may be accessed, as well as the types of the future objects. Section 3.2 presents an algorithm to identify methods that perform non-constant amount of work and are candidates for asynchronous execution, and Section 3.3 discusses how our transformation preserves the data dependences in the input program.

3.1 Inter-procedural Future Analysis

As indicated earlier, past work on future synthesis did not analyze the types of the future objects. Instead, they set the type of all future objects to `Object` in Java programs as in [27], or worked with untyped programs as in MultiLisp [15]. In contrast, our work attempts to determine the most precise type information for future objects as possible. We do so by using the IFDS [28] algorithm as the foundation for solving the future-analysis problem. IFDS can be used to compute the meet-over-all-valid-paths solution for all inter-procedural, finite, distributive subset problems. In the IFDS framework, the input program is represented as a directed graph called the *super graph*. The super graph consists of a collection of flow graphs, one for each procedure in the input program. The analysis is solved in polynomial time by reducing it to a graph-reachability problem.

The goal of the analysis is to find the set of variables and fields that *may/must* refer to a future object during all its accesses in the program, as well as the most precise type that can be identified statically for the future object, where a future object is the result of any expression of the form `async expr`. Our analysis finds the solution to two problems: *may-be-future* and *must-be-future*. The solution to the *must-be-future* problem is computed as the complement of the solution to the *may-not-be-future* problem. At any

given statement, $\mathcal{M}$ represents the set of variables and fields that *may-be-future* and $\mathcal{N}$ represents the set of variables and fields that *may-not-be-future*. The $\mathcal{M}$ and $\mathcal{N}$ sets need not be disjoint; in fact, the most conservative solution is to simply state that all variables and fields belong to both sets.

Normal flow functions are applied to all statements that contain neither call nor return statements. Examples of normal flow functions for computing $\mathcal{M}$ are given in Figure 5. An `async` expression generates a future object, whereas the result of a `new` expression cannot be a future object. The other three kinds of assignment statements may propagate a reference to a future object from the right-hand side to the left-hand side of the assignment. Our analysis builds on type-based alias analysis [9], and its precision can be improved by incorporating more complex alias analysis algorithms into the framework. However, more precise whole program alias analysis could be a scalability bottleneck for large applications. One drawback of using type-based alias analysis is that all elements of arrays of type τ in the program will be marked as `may-future` if any future object is stored into an array of type τ. We assume a universe Var of variable names, F of field names, $\mathcal{T}$ of class names, where $\mathtt{x},\mathtt{t},\mathtt{r},\mathtt{a}_0,\ldots\mathtt{a}_n,\mathtt{p}_0,\ldots\mathtt{p}_n \in Var$, $\mathtt{f} \in F$, and $\tau \in \mathcal{T}$.

Call flow functions handle the data flow from a method call statement into the called procedure. The context change from the body of the caller to the body of the callee is modeled by replacing references to actual parameters, $\mathtt{a_i}$ by references to formal parameters, $\mathtt{p_i}$. All fields that may be a future at the call site may also be a future at the start node of the callee. The call flow function at the call site, `c` is shown below, where $\mathtt{a_i} \xrightarrow{\mathtt{c}} \mathtt{p_i}$ represents the binding of the actual parameter $\mathtt{a_i}$ to the formal parameter $\mathtt{p_i}$.

$$\lambda Y.\{\forall \mathtt{i}, \mathtt{p_i} \mid \mathtt{a_i} \in Y \wedge \mathtt{a_i} \xrightarrow{\mathtt{c}} \mathtt{p_i}\} \cup \{\mathtt{f} \mid \mathtt{f} \in Y \wedge \mathtt{f} \in F\}$$

At a return statement, the data flow set at the callee is mapped back to the caller by the *return flow function*. The return value, `r` in the callee is mapped to the left hand side of the assignment in the caller. All fields that may be a future in the callee may also be a future at the call site. Return flow function at the call site, `c` is given below, where $\mathtt{r} \xrightarrow{\mathtt{c}} \mathtt{x}$ represents the binding of the return value, `r` in the callee to the variable `x` in the caller.

$$\lambda Y.\{\mathtt{x} \mid \mathtt{r} \in Y \wedge \mathtt{r} \xrightarrow{\mathtt{c}} \mathtt{x}\} \cup \{\mathtt{f} \mid \mathtt{f} \in Y \wedge \mathtt{f} \in F\}$$

A *call-to-return flow function* intra-procedurally propagates data flow values that are independent of the call. The call-to-return flow function for computing $\mathcal{M}$ is the identity function.

The *may-not-be-future* analysis is performed after *may-be-future* analysis. Examples of normal flow functions for *may-not-be-future* analysis are given in Figure 6. The main difference relative to *may-be-future* is in the functions for the `async` and the `new` expressions. The synthesis algorithm does not create a future task from an async expression, `async e`, if `e` may be a future object (thereby ensuring

Statement	Data flow function
`s: t := async e;`	$\lambda Y.(\text{if } \mathtt{e} \in \mathcal{M}(\mathtt{s}) \text{ then } Y \cup \{\mathtt{t}\} \text{ else } Y)$
`x := new` τ`;`	$\lambda Y.(Y \cup \{\mathtt{x}\})$
`x.f := t;`	$\lambda Y.(\text{if } \mathtt{t} \in Y \text{ then } Y \cup \{\mathtt{f}\} \text{ else } Y)$
`t := x.f;`	$\lambda Y.(\text{if } \mathtt{f} \in Y \text{ then } Y \cup \{\mathtt{t}\} \text{ else } Y)$
`x := t;`	$\lambda Y.(\text{if } \mathtt{t} \in Y \text{ then } Y \cup \{\mathtt{x}\} \text{ else } Y)$

Figure 6: Examples of normal flow function for computing $\mathcal{N}$ (may-not-be-future)

```
1  class TreeNode {
2    private future<TreeNode> left, right;
3    ...
4    TreeNode bottomUpTree(int item, int depth){
5      if (depth > 0) {
6        future<TreeNode> l = async<TreeNode> {
7          return bottomUpTree(2*item-1, depth-1);
8        }
9        future<TreeNode> r = async<TreeNode> {
10         return bottomUpTree(2*item, depth-1);
11       }
12       return new TreeNode(l, r, item);
13     } else {
14       return new TreeNode(item);
15     }
16   }
17   ...
18   int itemCheck() {
19     ...
20     future<Integer> li = async<Integer> {
21       return left.get().itemCheck();
22     };
23     future<Integer> ri = async<Integer> {
24       return right.get().itemCheck();
25     };
26     return item + li.get() - ri.get();
27   }
28 }
```

Figure 7: Binary Tree program after synthesis of futures

that no nested futures are created). Therefore the flow function for `s: t := async e;` checks that $\mathtt{e} \in \mathcal{M}(\mathtt{s})$, before adding `t` to $\mathcal{N}$, where $\mathcal{M}(\mathtt{s})$ denotes the set of variables and fields that may refer to a future immediately before statement `s`.

The result of future analysis are the sets $\mathcal{M}$ and $\mathcal{N}$, which will be available after the IFDS algorithm converges. The algorithm has worst-case complexity $O(ED^3)$, where E is the number of control-flow edges (or statements) of the analyzed program and D is the size of the analysis domain, where the domain consists of the set of all variables and fields in the program. We have not found this worst-case complexity to be a limitation in practice.

Appendix A contains the details of the transformation step based on the result of the data flow analysis. The program from Figure 4 after synthesis of futures is shown in Figure 7. The calls to `bottomupTree` and `itemCheck` are translated to future tasks. The types of the local variables `l` and `r` and fields `left` and `right` are changed to `future<TreeNode>`. The synthesis algorithm also inserts

`get` operations before the use of future objects in lines 20-26. Note that the synthesis algorithm does not insert `get` operations in line 12, where references to future objects are passed as arguments to the `TreeNode` constructor.

3.2 Candidate Future Identification

Our tool annotates calls to a subset of the pure methods identified by ReImInfer as async expressions. Although it is safe to execute all pure method calls as future tasks, it is not beneficial to execute method calls that perform insignificant work as separate tasks. Therefore, we identify methods that perform repetitive computations as candidates, by analyzing the call graph of the program and control flow graphs of each of the methods in the program.

Algorithm 1 Async method identification

Input: Call Graph of the Program , CFGs of each of the Methods
Output: Set of async methods A, Set of non-async methods S

```
1:  for each M ∈ {M_1, .., M_n} do
2:      if (ISLEAF(M) and not HASLOOPS(M)) or
3:        not ISPURE(M) or HASEXCEPTIONS(M) then
4:          S ← S ∪ {M}
5:      end if
6:      if (ISRECURSIVE(M) or HASLOOPS(M)) and
7:        ISPURE(M) and not HASEXCEPTIONS(M) then
8:          A ← A ∪ {M}
9:      end if
10: end for
11: Worklist ← {M_1, .., M_n} − S − A
12: while Worklist ≠ ∅ do
13:     M ← EXTRACT(Worklist)
14:     async ← False
15:     for each C ∈ CALLEES(M) do
16:         if C ∈ A then
17:             A ← A ∪ {M}
18:             async ← True
19:             break
20:         end if
21:     end for
22:     if async = False then
23:         S ← S ∪ {M}
24:     end if
25: end while
```

Algorithm 1 classifies the methods in the input program as `async methods` and `non-async methods`. A method is classified as an async method if it or any method that it calls contains repetitive structures in the form of loops or recursive cycles. This classification is refined later in the Parallelism Benefit Analysis (PBA) step. Lines 1-10 of Algorithm 1 initialize the sets A and S, which are the set of async methods and set of non-async methods respectively. S is initialized to contain all non-pure methods and leaf methods (methods which do not call other methods) with no loops in the method body. A is initialized to contain methods that are either recursive or contain a loop, with two further constraints: the method must be pure (ISPURE(M)), and must not throw any exceptions (HASEXCEPTIONS(M) is false).

Purity ensures that asynchronous execution of the method with a future result will not result in non-deterministic behavior (provided that any input variables that may be mutated after the method call are cloned, as discussed in Section 3.3). Exceptions represent a special kind of side effect that is typically not included in the scope of purity analysis. A common assumption in defining the semantics of exceptions with futures is to propagate any exception thrown by the asynchronous task at the point when the get() operation is performed. However that approach makes it challenging to execute `foo()` asynchronously in scenarios such as the following,

```
try { x = foo() ; } catch { } ; y = x.z;
```

in which any exception thrown by `foo()` in the sequential version will be "swallowed" before the result of `x` is accessed as `x.z`, while, under common assumptions, `x.get().z` could throw an exception in the parallel version. The **not** HASEXCEPTIONS(M) check ensures that this situation will not occur in our approach. While it is possible to identify some weaker sufficient conditions to be used as a replacement for **not** HASEXCEPTIONS(M), our experience has been that the **not** HASEXCEPTIONS(M) check provides a simple and effective means for ensuring correctness of our approach in the presence of exception semantics without limiting parallelism in practice.

The loop in lines 12-25 iteratively adds the remaining methods to S and A. The EXTRACT method extracts a method M from the worklist such that all the methods invoked by M are already classified as async or non-async. A method that calls an async method is added to A and a method that calls only non-async methods is added to S.

3.3 Preserving Data Dependences

The parallel program after future synthesis is data race free and deterministic if it preserves all data dependences in the input sequential program. Purity analysis ensures that asynchronous execution of the candidate future methods do not violate *flow* (Read after Write) and *output* (Write after Write) dependences, since pure methods do not mutate global data. To preserve *anti* (Write after Read) dependences in the input program, our algorithm copies (clones) all mutable data read by candidate futures and the future tasks performs all reads on the copied data. A weaker sufficient condition (which leads to less copying) is to copy all memory locations, L such that $L \in READ(F) \cap MOD(C)$, where F is the candidate future function, and C is the continuation after the call to F. $READ$ is computed by standard side-effect analysis, and MOD is computed by a backward data flow analysis on the inter-procedural control flow graph, where $READ$ and MOD represent the *read set* and *modification set* respectively.

4. Parallelism Benefit Analysis

Section 3 presented the analysis for synthesizing a parallel program, in which all pure method invocations are executed asynchronously. We refer to this program as a parallel program with *ideal parallelism*, which has the smallest possible critical path length (CPL) if we ignore all overheads of parallelism including those arising from task creation, task termination, and task synchronization. In practice, these operations can incur significant overhead, and it is necessary to ensure that every task has sufficient granularity to justify the task creation, task termination, and synchronization overheads, and there is parallelism benefit that arises from each task creation. We now present an algorithm to classify invocations of a pure method, M at call site c into one of the following three classes:

- ***Sequential:*** A method call is classified as sequential if all invocations of M at call site c must be executed sequentially.
- ***Parallel:*** A method call is classified as parallel if all invocations of M at call site c can be executed asynchronously.
- ***Conditional Parallel:*** A method call is classified as conditional parallel if a subset of the invocations of M at c must be executed asynchronously and the rest must be executed sequentially. In this case, the determination of whether a specific dynamic call should be executed sequentially or in parallel will be made by evaluating an automatically synthesized predicate expression at runtime.

The classification algorithm first constructs a data structure called the *weighted computation graph* (WCG), which is introduced in Section 4.1. Next, Section 4.2 presents an algorithm that uses the WCG to classify the calls into the three categories listed above. The WCG construction algorithm takes as input the parallel program with ideal parallelism synthesized by the algorithm in Section 3 and one or more test inputs for the program.

4.1 Weighted Computation Graph

A weighted computation graph (WCG) is a directed acyclic graph that is built at runtime to capture 1) the happens-before relationships among the step instances of a parallel program's execution, 2) the work done by each of the steps, and 3) the overheads incurred in task creation, task termination, and synchronization. Computations are represented in the WCG using step nodes, which are defined as follows:

Definition 1. A *step node* represents a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an async or a future get operation.

Definition 2. The weighted computation graph (WCG) for a given execution is a directed acyclic graph with five different types of nodes:

- A step node, S_n represents a sequential computation. The weight of step node is the total number of instructions executed to complete the step.
- A spawn node F_n represents the creation of child task. The weight of spawn node T_{spawn} represents the overhead in the parent task for task creation. This weight includes the overhead of copying the mutable data read by the child task.
- A join node J_n represents the join operation with another task. The weight of join node T_{join} represents the overhead of a join operation in the waiter task.
- A start node B_n is the first step in a task. The weight of start node T_{start} represents the overhead of creating and scheduling a task.
- An end node E_n is the last step in a task. The weight of end node T_{end} represents the overhead of task termination.

Next, we discuss how to build the WCG during program execution. We first instrument the parallel program with ideal parallelism generated by the algorithm in Section 3. The instrumented program is then executed on a test input in serial, depth-first order (like a sequential Java program) to construct the WCG. The instrumented code performs the WCG construction as follows: When the main task starts execution, the WCG will contain two nodes: 1) B_1 which corresponds to the start node of the main task and 2) step node S_1 which corresponds to the starting computation inside main. The edge $B_1 \rightarrow S_1$ represents the ordering between B_1 and S_1.

Future Task Creation When a task T_a creates a child task T_b, a spawn node F_i is created and an edge is inserted from S_j to F_i, where S_j is the step immediately preceding the spawn operation in T_a. A start node B_k corresponding to T_b is created and an edge is inserted from F_i to B_k. Task T_b is now executed, and the next node N (step, spawn, join, or end node) is added as a successor of B_k.

Future Task Termination When a task T_b completes execution, an end node E_i is created and an edge is inserted from S_j to E_i, where S_j is the last step in T_b. The program execution now continues in T_a which is the parent task of T_b. The next node to be added is the successor of F_k which is the spawn node in T_a corresponding to the creation of T_b.

Future Join When task T_a performs a join operation on task T_b, a join node J_i is created, and an edge is inserted from S_j to J_i, where S_j is the step immediately preceding the join operation in T_a. An edge is inserted from E_k to J_i, where E_k is the end node in T_b. Execution of T_a continues at node N, which is the successor of J_i.

Example The WCG for the Binary Tree program in Figure 7 for input `depth=2` is shown in Figure 8. The weights of each of the nodes are shown above the node. Task T_1 which consists of the directed path from B_1 to E_1 is the main task. T_2, T_3, T_4, T_5, T_6, and T_7 are future tasks created in

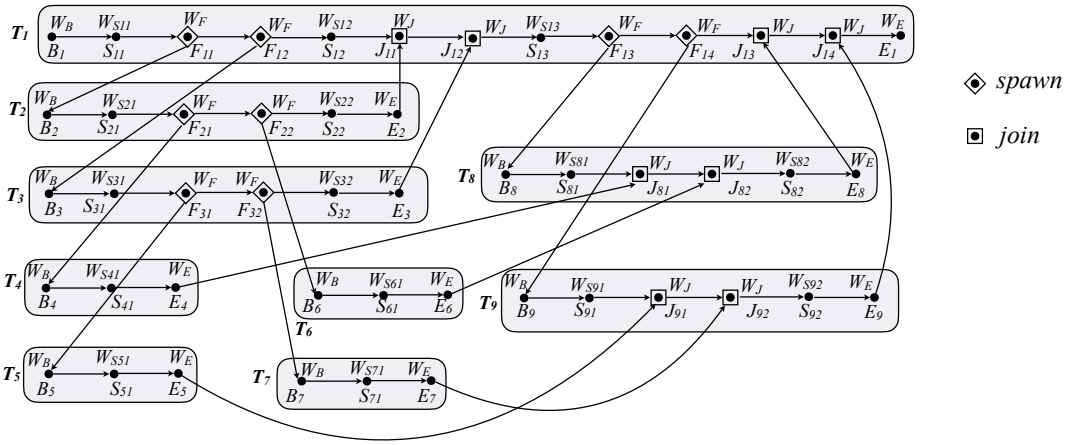

Figure 8: Weighted computation graph for Binary Tree program in Figure 7 for `depth=2`

line 6 and line 9 for the invocations of `bottomupTree`. T_8 and T_9 are future tasks created in line 20 and line 23 for the invocation of `itemCheck`. This WCG demonstrates the generality of synchronization patterns possible with futures. For instance, the edge from E_7 to J_{92} (T_7 to T_9) due to the future `get` operation is not possible in more structured fork-join models.

4.2 Classification of Pure Function Calls

We now analyze the WCG to identify tasks that give no benefit from asynchronous execution. Based on this analysis, we classify pure method invocations as parallel, sequential or conditional parallel. For every task T_a, our analysis tries to answer two questions: 1) Is the work done by task T_a of sufficiently coarse granularity to justify the task creation, task termination, and synchronization overhead? and 2) Is there sufficient work that can be overlapped with the execution of T_a? We use the critical path length (CPL) as the cost metric for evaluating the profitability of parallelization, where the critical path is defined as follows.

Definition 3. The critical path of a weighted computation graph is the longest weighted path in the WCG, where the weight of a path is the sum of the weights of all the nodes included in the path.

Algorithm 2 presents our approach for evaluating the parallelism benefit for each of the tasks in the WCG. The algorithm takes as input the WCG, G and the set of all tasks, T. The outputs of the algorithm are the set of parallel tasks, P and the set of serial tasks, S. The algorithm uses a greedy strategy, evaluating the tasks in bottom-up, right-to-left order. The algorithm merges a task with its parent, if executing that particular task asynchronously does not yield any benefit. The bottom-up approach ensures that the tasks of smaller granularity are first considered as merge candidates. Evaluating the right siblings of a task T_a as merge candidates before T_a itself ensures that the algorithm obtains a more

Algorithm 2 Parallelism benefit analysis

Input: Computation graph G, Set of tasks T
Output: Set of sequential tasks S, Set of parallel tasks P

for each $t \in T$ **do**
  if WORK$(t) < T_{spawn}$ **then**
    $S \leftarrow S \cup \{t\}$
    $G \leftarrow$ MERGEPARENT(G, t)
    $Visited \leftarrow Visited \cup \{t\}$
  end if
end for
for each $t \in T - Visited$ **do**
  $P \leftarrow P \cup \{t\}$
  if CHILDREN$(t) = \emptyset$ **and** RIGHTSIBLINGS$(t) = \emptyset$ **then**
    $Worklist \leftarrow Worklist \cup \{t\}$
  end if
end for
$CPL \leftarrow$ LONGESTPATH(G)
while $Worklist \neq \emptyset$ **do**
  Remove t from $Worklist$
  $Visited \leftarrow Visited \cup \{t\}$
  $G' \leftarrow$ MERGEPARENT(G, t)
  $CPL' \leftarrow$ LONGESTPATH(G')
  if $CPL' < CPL$ **then**
    $G \leftarrow G'$
    $P \leftarrow P - \{t\}$
    $S \leftarrow S \cup \{t\}$
    $CPL \leftarrow CPL'$
  end if
  for each $t_1 \in T - Visited$ **do**
    if (CHILDREN$(t_1) \cup$ RIGHTSIBLINGS(t_1))
      $\cap Visited = \emptyset$ **then**
      $Worklist \leftarrow Worklist \cup \{t_1\}$
    end if
  end for
end while

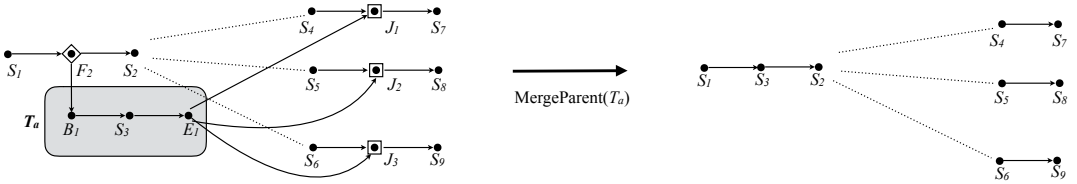

Figure 9: Merge Parent transformation on the computation graph, where the task T_a is merged with its parent.

accurate estimate of the total work in the continuation before making a decision on the profitability of executing T_a asynchronously. Lines 1-7 of Algorithm 2 classify a task as serial and merges it with its parent if the total work done by the task is less than T_{spawn}. Lines 8-13 initialize the worklist with the set of tasks which have no children and no right siblings. The set of parallel tasks P is initialized to contain all non-serial tasks. Lines 15-32 remove a task t from the worklist, classifies it as serial/parallel and updates the worklist with tasks which are ready to be evaluated. Lines 18-25 merge t with its parent and checks if the CPL after merging is smaller than the CPL before merging. If the merging results in a smaller CPL, the task t is classified as serial, and the WCG is updated.

Figure 9 shows an example of the MERGEPARENT transformation on the WCG, where the task T_a is merged with its parent. F_2 is the spawn node corresponding to the task T_a in the parent task. Task T_a consists of the step node S_3 in addition to the start node and the end node. There are three separate join nodes corresponding to T_a, which are J_1, J_2, and J_3. The dotted edges from S_2 to S_4, S_5, and S_6 represent paths in the WCG. The MERGEPARENT transformation removed the spawn, start, end, and join nodes corresponding to T_a and inserted S_3 along the path from S_1 to S_2 where S_1 and S_2 are the nodes preceding and succeeding the spawn node F_2 in the WCG. This transformation ensures that there are paths from S_3 to S_7, S_8, and S_9 which are successor nodes of the join nodes in the input WCG.

Our approach performs parallelism benefit analysis separately on each of the WCGs corresponding to each of the test inputs. The output of parallelism benefit analysis is the set of serial and parallel tasks. Next, we merge the output of parallelism benefit analysis for all WCGs at a particular call site and identify parallel and conditional parallel call sites. Classifying call sites as serial, parallel, and conditional parallel based on this output is straightforward. If all instances of a method at a particular call site are serial/parallel, then that call site is classified as serial/parallel. If only a subset of instances of a method at a particular call site are serial, we mark it as conditional parallel.

5. Threshold Expression Synthesis

Parallelism benefit analysis classifies the pure method calls as sequential, parallel, and conditional parallel. Conditional parallel calls are method calls that should be executed asynchronously only if the work done by the method is greater than the *sequential threshold*, which is the minimum amount of work (in terms of instruction count) that must be done by a task to justify the task overhead. For a method invocation `obj.f(`a_1, a_n`)`, which is classified as conditional parallel, our goal is to generate the code shown in Figure 10. The conditional expression `cond` determines if the work done by `f` is greater than the sequential threshold and we refer to this as the *threshold expression*.

```
1 mayfuture<T> x;
2 if (cond)
3   x = async<T> { return obj.f(a1,..,an);};
4 else
5   x = obj.f(a1,..,an);
```

Figure 10: Conditional parallel execution of method call `obj.f(`a_1, a_n`)`

The work done by a pure method call `obj.f(`a_1, a_n`)` typically depends on 1) the type of `obj`, because a class hierarchy can contain multiple implementations for the same method and the type of `obj` determines which implementation of `f` is invoked, 2) the values of the arguments of `f`, and 3) the values of the fields of `obj`. For example, the total work done by the recursive Fibonacci function `int fib(int n)` depends on `n` and the work done by a sort function `int[] sort(int[] A)` depends on `A.length`. Here `n` and `A.length` represent the problem sizes for the two functions respectively.

While instrumenting the input program for WCG construction, we also instrument it to collect the following information for each invocation of a method `f` executed as a future task:

- The type `T` of the receiver object `X`
- The problem size parameters $p_1, .., p_k$ of `f`, which consists of
  - The values of numeric type arguments of `f`
  - The values of numeric type fields of arguments of `f`
  - The values of numeric type fields of `X`, where `X` is the receiver object of `f`
  - The size of collection/data structure (List, Array, String) type arguments of `f`

- The cumulative work, w done by `f` and all the methods that it calls. The work is computed in terms of the number of instructions executed.

Since `f` is a pure method, the threshold expression for `f` is usually a function of the problem size parameters and the type of the receiver object.

At each conditional parallel call site, the profile information is divided based on the type of the receiver object. Note that the profile information for each conditional parallel call site from multiple test inputs is merged before threshold expression synthesis. For each receiver type `T`, we construct a matrix, M with $k+1$ columns, where columns 1 to k contains the values of the problem size parameters p_1 to p_k, and column $k+1$ contains work w. For most pure methods, the problem size is one of $p_1, .., p_k$. In other cases, the problem size is an expression involving two or more of the parameters $p_1, .., p_k$. Our approach finds the threshold expression by performing a search on the space of expressions formed from $p_1, .., p_k$ and a set of arithmetic operators. Our current implementation handles arithmetic operators +, -, `min` and `max`. The search algorithm looks for an expression e, such that e has a monotonic relationship with w – as the value of e increases, so does the value of w. We use Spearman's rank correlation [30] coefficient, ρ as the metric for monotonicity. Spearman's correlation coefficient assesses how well the relationship between two variables X and Y can be described using a monotonic function. Note that ρ is a non-parametric measure and is not based on a possible relationship of a parameterized form (such as a linear relationship). The value of ρ lies between +1 and -1 inclusive, where 1 is a total positive correlation, 0 is no correlation, and -1 is a total negative correlation. Our algorithm starts by computing ρ between each of the problem size parameters p_i and the work w. If the algorithm finds a p_i which has high rank correlation to w ($\rho(p_i, w) > \rho_{threshold}$, where $\rho_{threshold} = 0.9$), the algorithm terminates returning p_i as the problem size. If none of the parameters has high correlation with w, the algorithm computes the correlation of expressions involving two parameters such as $p_i + p_j$ and $p_i - p_j$. This is done by constructing a new matrix in which each of the columns corresponds to an expression. The search continues with larger expressions until an expression with the desired correlation is found or when the algorithm has explored all expressions of size n, where n is a tuning parameter for the search. This search algorithm for an expression with syntactic constraints which meets a correctness specification (high correlation) is similar in spirit to syntax guided synthesis [1]. If TES for a call site c is unable to find a threshold expression, we classify c as parallel/serial based on the frequency of parallel/serial invocations in the output of parallelism benefit analysis. Next, we find the minimum value, v of e for which the method must be executed asynchronously by a lookup of M. This information on which rows in M corresponds to parallel execution is available from parallelism benefit analysis. The result of threshold expression synthesis for a given receiver type `T` is the expression ((`obj instanceof T`) $\wedge$ ($e \leq v$)) or ((`obj instanceof T`) $\wedge$ ($e \geq v$)), depending on whether the correlation between the expression and the work is positive or negative. The final threshold expression is a disjunction of threshold expressions for each of the receiver types. The instanceof check can be eliminated if the declared type of the receiver object is same as `T`. Appendix B presents the full algorithm for threshold expression synthesis.

As an example, consider the `int[] mergesort(int a[], int start, int end)` function, which sorts the elements of array `a[]` starting at index `start` and ending at index `end`. The problem size parameters for this function are 1) `a.length`, which is the length of the array `a`, 2) `start` and 3) `end`. The TES algorithm computes the rank correlation between each of these parameters and the work w did by the function. The algorithm continues the search since none of these parameters have a high correlation with the work. Next, the algorithm computes the correlation between expressions involving two parameters such as `a.length + start`, `start + end`, and `end - start`. Finally, the algorithm returns `end - start` as the threshold expression, since it has a high correlation to the work done by the function.

6. Final Future Synthesis

The last step in our parallelization tool is the generation of parallel code, in which pure method calls which are found to be beneficial by the analysis in Section 4 are executed asynchronously (and the others are executed sequentially). The inputs to the parallel code generation are the input sequential program, the set of parallel call sites, the set of conditional parallel call sites and the threshold expressions for each of the conditional parallel call sites computed by the algorithm in Section 5. (Note that this step uses the original sequential program as input, and not the parallel program from Section 3.)

The final future synthesis step includes the following steps:

1. Generate conditional statements at conditional parallel call sites using threshold expressions. The true branch represents the case where the work done by the method is greater than the sequential threshold and the false branch represents the case where the work done by the method is less than the sequential threshold. We annotate the method call in the true branch as an async expression.
2. Annotate all parallel call sites as async expressions
3. Clone all inputs to each pure method that may be modified in any of the continuations that follow each of the parallel and conditional parallel calls, and replace all references to those inputs in the pure method by references to the cloned data.

4. Synthesize futures in the async-annotated program resulting from the previous steps, using the synthesis algorithm presented in Section 3.1 (with additional details in Appendix A).

As discussed earlier, the result of final future synthesis for the program in Figure 1 can be seen in Figure 2.

7. Experimental Evaluation

7.1 Experimental Setup

In this section, we summarize the implementation and setup used in our experimental evaluation. The different components of our system were implemented in Habanero Java (HJ) [6] compiler and runtime as follows. The HJ compiler extends the Soot framework [35] for bytecode transformations. The inter-procedural future analysis is implemented as a new compiler analysis pass in Heros [3], a scalable, highly multi-threaded implementation of the IFDS framework, which can be invoked from Soot. Our inter-procedural analyses used the call graph provided by the Soot framework, which is obtained through class hierarchy analysis and is sound. The insertion of future operations is implemented as a subsequent compiler transformation pass that updates Soot's Jimple [35] intermediate representation extended for HJ programs. Instrumentation of programs for WCG construction and computation of instruction counts of steps is also implemented as a bytecode-level transformation pass on Jimple. The instrumentation pass inserts callbacks to the HJ runtime at all future task creation, task termination, and synchronization points in the program and also inserts counters to compute the dynamic number of instructions executed during the program execution. For each method call that is executed as a future task, we instrument the bytecode to collect the values of the arguments of the method, the type and the fields of the receiver object. This information is used for threshold expression synthesis. The instrumented program, during execution, writes the profile information to a file. Parallelism benefit analysis is implemented as a compiler pass in the HJ compiler which reads the profile information, analyzes it and finds the set of method calls that are parallel and conditional parallel. We used the JGF ForkJoin microbenchmark to measure the overheads of task creation and task termination for our runtime and a given hardware platform. This information and the profile information is passed to threshold expression synthesis, which determines the threshold expressions for each of the conditional parallel method calls. Finally, the parallel code is generated by invoking the inter-procedural future analysis and the future transformation pass.

Our experiments were conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection, and other JVM services, we report the steady state mean execution time of 30 runs repeated in the same JVM instance for each data point. In all our measurements, we only used 8 of the 16 cores to execute the application (by using HJ's "-places 1:8" option), so as to further reduce the impact of system perturbations.

```
1 Integer f1 = fib(n-1);
2 Integer f2 = fib(n-2);
3 return f1+f2;
```

Figure 11: Sequential recursive Fibonacci invocation

We evaluated the parallelization tool on a suite of nine benchmarks listed in Table 1. Our approach targets applications in which pure method calls perform a significant amount of work. Therefore, we chose benchmarks in which at least 50% of the sequential work is performed in pure method calls. We did this by using ReImInfer to identify pure methods and by inserting timers around calls to pure methods. The fourth column of Table 1 shows the input size used for profiling the program ("Train"). The fifth column shows the input size used for performance evaluation of the parallelized programs ("Ref").

7.2 Experimental Results

We now present experimental results for our automatic parallelization approach. Table 2 shows the results of parallelism benefit analysis. The second column shows the number of call sites that were identified as candidates for future task creation by the algorithm in Section 3.2. The third, fourth, and fifth columns show the number of call sites that were identified as serial, parallel, and conditional parallel respectively by parallelism benefit analysis.

A call site may be classified as serial if there is insufficient work done by the method or if there is insufficient parallelism. For example, in the Fibonacci program in Figure 11, the call to `fib` in line 2 will be classified as serial, since there is no benefit in executing it as a separate future task, whereas the call in line 1 will be classified as conditional parallel, because 1) the work done by the function depends on the value of `n` and 2) if the call is executed as a future task, it can execute in parallel with the call in line 2.

Table 2 shows that a subset of methods identified by the algorithm in Section 3.2 benefit from asynchronous execution, thereby reinforcing the importance of parallelism benefit analysis in choosing method calls for parallelization.

Table 3 shows the statistics resulting from the final future synthesis step, which is performed after parallelism benefit analysis and threshold expression synthesis. It shows the number of variable types that were changed, and the number of `get()`, `instanceof`, and cast operations that were inserted by future synthesis. Overall, these statistics indicate that significant programmer effort is required to manually parallelize a sequential program using futures and that this effort may need to repeat for different platforms (due to differences in overheads and available hardware parallelism).

Source	Benchmark	Description	Input Size (Train)	Input Size (Ref)
JGF [4]	Series	Fourier coefficient analysis	size A	size B
SPECjvm2008 [31]	MPEGaudio	MPEG audio decoder	4 mp3 files	12 mp3 files
CLBG [23]	Binary Tree	Tree construction traversal	$depth = 14$	$depth = 20$
Jolden [5]	TreeAdd	Recursive depth-first traversal of a tree	$depth = 15$	$depth = 24$
BOTS	Nqueens	N Queens problem	$n = 9$	$n = 13$
HJ Bench	Fibonacci	Compute *n*th Fibonacci number	$n = 22$	$n = 38$
	MatrixEval	Matrix expression evaluation	200x200	500x500
	Mergesort	Mergesort	$n = 16000$	$n = 1000000$
	Quicksort	Quicksort	$n = 10000$	$n = 1000000$

Table 1: List of benchmarks evaluated. **Input Size(Train)** is the input size used for profiling the parallel program for parallelism benefit analysis and **Input Size(Ref)** is the input size used for performance evaluation.

Benchmark	#Candidate	#Serial	#Parallel	#Conditional
Series	3	1	2	0
MPEGaudio	1	0	1	0
Binary Tree	12	9	1	2
TreeAdd	6	4	0	2
Fibonacci	3	2	0	1
MatrixEval	3	2	1	0
Mergesort	4	3	0	1
Nqueens	3	2	0	1
Quicksort	3	2	0	1

Table 2: Number of call sites identified as serial, parallel, and conditional parallel by parallelism benefit analysis

Table 4 compares the execution times of the sequential and automatically parallelized versions of each program in our benchmark suite. All programs were run with 16GB heap space. **Par(No PBA)** shows the parallel execution time without parallelism benefit analysis. All call sites that were identified as candidates for future task creation by the algorithm in Section 3.2 were executed as parallel tasks with no threshold expressions. OOM (Out Of Memory) represents cases where execution did not complete because of insufficient heap memory. **Par(No TES)** shows the parallel execution times with parallelism benefit analysis but without threshold expression synthesis. In this case, we used the parallel/serial frequency information at a call site to classify a method call as either serial or parallel. There are no conditional parallel method calls in this case. **Par** is the parallel execution time with parallelism benefit analysis and threshold expression synthesis. All the applications showed significant performance improvements with our approach.

Table 5 compares the execution times of the parallel versions of each program with conditional parallel sites when using different threshold values. **Threshold** shows the parallel execution time when using threshold values computed by our threshold expression synthesis algorithm. For this sensitivity analysis, we used two threshold values which allow higher parallelism and two threshold values which allow lower parallelism compared to the threshold value computed by threshold expression synthesis. **Threshold - 1** shows the parallel execution time when a threshold value that allows higher parallelism compared to **Threshold** is used. **Threshold - 2** shows the parallel execution time where the threshold value allows higher parallelism compared to **Threshold - 1**. Similarly, **Threshold + 1** and **Threshold + 2** shows the parallel execution times where the parallelism is lower compared to **Threshold**. For example, the threshold value computed by TES for the Binary Tree benchmark is 12. Therefore, we used threshold values of 10 (**Threshold - 2**), 11 (**Threshold - 1**), 12 (**Threshold**), 13 (**Threshold + 1**) and 14 (**Threshold + 2**) for the sensitivity analysis. For Quicksort and Mergesort benchmarks, the threshold values are varied by a factor of two. The results indicate that the performance of the parallel program when using threshold values computed by threshold expression synthesis is better or comparable to the performance of the program when using a different threshold value.

In summary, these results indicate the importance of parallelism benefit analysis and threshold expression synthesis in automatic generation of task parallelism.

8. Related Work

8.1 Futures

Futures were introduced by Halstead as an explicit concurrency primitive for functional programming in Multilisp [15], an untyped language that did not need the synthesis capabilities introduced in our work. Flanagan and Felleisen [13] defined a whole program analysis to reduce runtime checks for futures in dynamically typed languages. In contrast, our work synthesizes future operations, synchronization, and runtime checks, while also providing paral-

Benchmark	#Future	#MayFuture	#Gets	#Instanceof	#Typecasts
Series	2	3	2	4	4
MPEGaudio	3	0	1	0	1
Binary Tree	2	15	4	6	7
TreeAdd	0	10	4	8	8
Fibonacci	0	2	1	2	2
MatrixEval	8	0	3	0	3
Mergesort	0	4	1	2	2
Nqueens	0	4	2	4	4
Quicksort	0	2	1	2	2

Table 3: Synthesis statistics. **#Future** gives the number of variables and fields whose type got changed to `future<T>`, **#MayFuture** gives the number of variables and fields whose type got changed to `mayfuture<T>`. **#Gets** , **#Instanceof** and **#Typecasts** are the number of `get()`, `instanceof` and cast operations inserted by future synthesis.

Benchmark	Seq	Par(No PBA)	Par(No TES)	Par	Speedup (Seq/Par)
Series	25,973.35	4,175.49	4,203.84	4,201.22	6.18
MPEGaudio	6,691.52	2,839.76	2,838.51	2,835.21	2.36
Binary Tree	527.50	167,674.52	529.11	271.33	1.94
TreeAdd	586.54	293,431.26	602.21	259.49	2.26
Fibonacci	473.92	OOM	474.33	64.07	7.40
Quicksort	282.71	OOM	296.39	70.58	4.00
MatrixEval	1,853.83	884.11	359.23	358.86	5.17
Mergesort	151.31	OOM	151.57	40.96	3.69
Nqueens	4,242.57	OOM	4,211.21	1,199.07	3.54
GeoMean	-	-	-	-	3.69

Table 4: Comparison of execution times in milliseconds of the sequential and parallel versions of the program. **Seq** is the sequential execution time, **Par(No PBA)** is parallel execution time without parallelism benefit analysis, **Par(No TES)** is parallel execution time without threshold expression synthesis and **Par** is the parallel execution time of the final program generated by our approach. Parallel versions were run on 8 cores. OOM (Out Of Memory) represents cases where execution did not complete because of insufficient heap memory.

Benchmark	Threshold - 2	Threshold - 1	Threshold	Threshold + 1	Threshold + 2
Binary Tree	309.18	292.13	271.33	295.85	306.62
TreeAdd	349.83	279.53	259.49	259.26	256.93
Fibonacci	OOM	88.2	64.07	57.71	54.37
Quicksort	89.74	85.18	70.58	66.69	69.73
Mergesort	43.92	40.68	40.96	41.79	41.76
Nqueens	1,461.63	1,261.53	1,199.07	1,556.3	4,665.8

Table 5: Comparison of execution times in milliseconds of the parallel versions of the programs with different threshold values for conditional parallel call sites. **Threshold** is the execution time with threshold expressions computed by TES. **Threshold - 1** and **Threshold - 2** are execution times of parallel programs with higher parallelism compared to **Threshold**. **Threshold + 1** and **Threshold + 2** are execution times of parallel programs with lower parallelism compared to **Threshold**.

lelism benefit analysis and threshold expression synthesis capabilities.

Safe futures [36] implement futures as software transactions so that safety violations (data races) can be avoided or corrected. Their approach requires a heavyweight runtime which supports object versioning, operation logging and other metadata. Performance results can vary widely depending on the number of safety violations detected at runtime. Navabi et al. [25] use static analysis to insert barriers, which preserve sequential semantics with the help of a lightweight runtime. Swaine et al. [33] provide a way to add parallelism to legacy runtime systems using futures. These approaches require the programmer to parallelize the program, and the framework handles conflicts due to shared data accesses. In contrast, our approach is fully automatic, targets pure method calls and has to deal with only anti dependences on mutable data, which are preserved by copying the data.

Pratikakis et al. [27] present a framework for transparently executing programs with asynchronous calls. They employ a static analysis based on qualifier inference to identify the proxy variables in the program. Their analysis is flow-insensitive and context-insensitive and does not differentiate *maybe proxy* variables from *mustbe proxy* variables. Due to these differences, their approach changes the types of all variables which could potentially be a proxy to `java.lang.Object`. Their framework also requires a type cast and an `instanceof` check at every potential proxy access. Their framework requires the programmer to annotate the async expressions, which can cause data races. The programmer also has to determine whether the annotated expressions would benefit from asynchronous execution.

Harris and Singh [16] presented a profile based parallelization for Haskell programs. Their approach selects thunks for parallel execution which are likely to be needed by the program and will run long enough to compensate the overheads. In contrast to our work, their approach does not require future synthesis and does not find threshold expressions which can dynamically determine if the work is of sufficient granularity to justify task creation overhead.

Directive-based Lazy Futures [38] require users to annotate declarations of all variables that store the return value from a function that can be potentially executed as futures with @future directives. Their approach does not allow the propagation of future objects across method boundaries, which can limit parallelism in many cases. In contrast, our approach automatically identifies the variables and fields which may hold references to future objects and inserts `get()`, without limiting the parallelism at method boundaries. Zhang et al. [39] and Navabi et al. [24] presented approaches for precise exceptions in the presence of futures, which is orthogonal to our work which addresses automatic parallelization.

8.2 Parallelism and Performance Profiling

Past work has used profile information and critical path analysis to analyze the parallelism in a given application. Kulkarni et al. [20] used a critical path based analysis to bring insight into the parallelism inherent in irregular algorithms that exhibit amorphous data parallelism. Kremlin [14], given a serial version of a program, will make recommendations to the user as to what regions (e.g. loops or functions) of the program to parallelize first using a hierarchical critical path analysis. It also provides a ranked order of specific regions to the programmer that are likely to have the largest performance impact when parallelized. Cilkview [17] scalability analyzer is a software tool for profiling and estimating scalability of parallel Cilk++ applications. It monitors the logical parallelism during an instrumented execution of the application on a single core. As Cilkview executes, it analyzes logical dependencies within the computation to determine its work and critical path length. It uses these metrics to estimate parallelism and predict the scalability of the application. Unlike Cilkview, which analyzes only the whole-program scalability of a Cilk computation, Cilkprof [29] collects work and critical-path length for each call site in the computation to assess how much each call site contributes to the overall work and span.

Alchemist [40] presents a profiling technique to automatically detect available concurrency in C programs. The profiler detects dependences between a construct (a loop, a procedure, or a conditional statement) and its continuation. The dependence distances between program points are then used to measure the effectiveness of parallelizing the construct, as well as identifying the transformations necessary to facilitate the parallelization.

Threshold expression synthesis finds an expression involving method arguments and receiver object fields, which is monotonic with respect to the work done by the function. Past work has tried to model the performance of programs as a function of the size of the input. Emilio Coppa et al. in [7] presented input-sensitive profiling, a method for automatically measuring how the performance of individual routines scales as a function of the size of the input. The key feature of their method is the ability to automatically measure the size of the input given to a generic code fragment. The tool estimates the input size by using the amount of distinct memory first accessed by a routine or its descendants as reads. This work was extended in [8] which takes into account dynamic workloads produced by memory stores performed by other threads and by the OS kernel. As noted in [7], their approach fails to characterize pure functional computations such as Fibonacci where the running time (or work) is determined by the values of one or more arguments. Algorithmic profiling [37] is an approach to automatically infer approximations of the expected algorithmic cost functions of algorithm implementations. Our method of determining the inputs to a function is similar to their approach. Their imple-

mentation automatically infers the inputs to the program, but the fitting of cost functions is done by hand. In contrast, our approach does not try to find an exact cost function, but uses a search algorithm to find an expression which is monotonic with respect to the work done by the method.

Duran et al. [10] presented a runtime technique to adaptively coalesce OpenMP tasks by employing a dynamic profiler. The profiler estimates the work performed by a task as the average work performed by all previously profiled tasks at that particular level/depth of the spawn tree. The work estimation does not depend on the computation performed by the task or the arguments to the computation, whereas our approach computes a distinct threshold expression for every call site and takes into account the arguments to the computation.

Thoman et al. [34] presented a combined compiler and runtime approach that enables automatic granularity control. They generate multiple versions of a given task of increasing granularity by task unrolling at compile time, and the runtime system selects a task version by estimating task demand. The number of generated versions depend on the granularity of the initial tasks, but the paper does not discuss how the task granularity is estimated. A runtime estimate of task demand could be combined with our approach to prevent task creation if the demand is low.

9. Conclusions

We presented a novel approach for automatically parallelizing pure method calls by using futures as the primary parallel construct. Given a sequential program, our algorithm automatically generates a parallel program in which pure method calls that benefit from asynchronous execution are executed as future tasks. Our approach addresses the major drawbacks of manually parallelizing programs using futures. Section 3 contains our algorithm for synthesizing future tasks and their associated type declarations with more precision than in past work. Section 4 describes our approach to classifying each pure method call as sequential, parallel, or conditional parallel, based on computing critical path lengths in a weighted computation graph that takes task creation, termination, and synchronization overheads into account. Section 5 contains our algorithm for synthesizing threshold expressions that can be evaluated at runtime, to ensure that a future task is only created when it is profitable to do so. We implemented all three steps in our approach, and evaluated the complete tool chain on a range of applications written in Java. When using 8 processor cores, the evaluation shows that our approach can provide significant parallel speedups of up to 7.4× (geometric mean of 3.69×) for sequential programs with zero programmer effort, beyond providing test cases for parallelism benefit analysis.

There are many opportunities for future research to build on the results of this paper. Future synthesis is interprocedural in scope and requires whole-program static analysis in general. A direction for future work is to make our approach modular by ensuring that there is no asynchronous information flow through future objects across components. Enhancements in alias analysis could further increase the precision of type information in the synthesized program. Alternatively, our approach can be applied to dynamically typed languages for which no type declarations need to be generated. There is also room to further study the impact of exception semantics on automatic parallelization with futures, and to explore the use of runtime checks for potential exceptions in candidate future tasks. There is a promising opportunity for code motion to separate future task creation, and the corresponding future `get()` operations as far as possible, so as to increase the parallelism in the program (akin to global instruction scheduling). Yet another direction for future work is to extend our approach so that it can be applied to programs with explicit task parallelism, thereby using future tasks to further increase parallelism; it would also be interesting to perform parallelism benefit analysis and threshold expression synthesis for explicitly-parallel programs so as to aid the programmer in granularity control of their parallelism. Finally, it would be desirable (albeit challenging) to perform parallelism benefit analysis and threshold expression synthesis at runtime, so that they can be better tuned to the underlying platform.

Acknowledgments

We are grateful to the authors of the ReImInfer tool [18], Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst for sharing their implementation of ReImInfer, and for answering our questions related to ReImInfer. We thank Swarat Chaudhuri and John Mellor-Crummey from Rice University for their feedback and suggestions. We also thank members of the Habanero group at Rice for valuable discussions related to this work and contributions to the Habanero Java infrastructure used in this research. We are grateful to the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF award 1302570.

A. Future Transformation

In this section, we present transformation rules for synthesizing futures based on the result of inter-procedural analysis from section 3.1. The transformation rules are shown in Table 6. The third column shows the input code, and the fourth column shows the transformed code if the conditions in the second column hold true. $\mathcal{M}(\mathtt{s})$ denotes the set of variables and fields which may refer to a future immediately before statement `s`. Similarly, $\mathcal{N}(\mathtt{s})$ denotes the set of variables and fields which may not refer to a future immediately before statement `s`. Rule 1 translates an `async` expression `async e` to a future task creation expression, if `e` may not be a future. Rule 2 changes the type of a variable from `T` to `future<T>`, if the variable must refer to a future object. Rule 3 changes the type of a variable to `mayfuture<T>`, if it may hold a reference to a future and a non-future object, where `T` is an application class. If `T` is a library class, the type is changed to `Object`. Rules 4-7 handle the different cases, where the object field, `f` is accessed. Rule 4 and 6 handle the case, where `x` must refer to a future object by inserting an `x.get()` operation which obtains the result of the future task. Rule 5 handles the case where `x` may not refer to a future object at statement `s`, but may refer to a future object at a different statement $\mathtt{s}_1$. In this case, a cast operation from `mayfuture<T>` to `T` is required before the field access at statement `s`. Rule 7 is the most general case, where `x` may refer to a future object or a non-future object at statement `s`. In this case, a runtime check is inserted which handles both the possible scenarios.

Our implementation also changes method parameter types and return types based on the result of the future analysis. When the parameter type (or return type) of a class member function is changed, the type declaration of the corresponding function in the super class is also updated. For instance, let $D_1, .., D_n$ be the subclasses of C and let $T_1, ..., T_n$ be the inferred type of the ith parameter of member function F in $D_1, .., D_n$ respectively. The transformation algorithm then updates the type of ith parameter of F in C to $lub(T_1, ..., T_n)$ which is the least upper bound type of the types $T_1..T_n$, where for any given type `T`, `T` $\leq$ `mayfuture<T>` and `future<T>` $\leq$ `mayfuture<T>`.

B. Threshold Expression Synthesis Algorithm

Algorithm 3 presents the high level view of our approach for threshold expression synthesis for a given call site `obj.f(`$\mathrm{a}_1..\mathrm{a}_n$`)`. Line 1 of the algorithm initializes the threshold expression to false. The loop in lines 2-22 iterates through each of the possible types of the receiver object and finds the threshold expression. Lines 3-18 handle the case where the call site is conditional parallel for type `T`. CONSTRUCTMATRIX constructs a matrix with m columns, where columns 1 to $m-1$ contains the data corresponding to the expressions and column m contains the work. MAXSPEARMANCORR computes the Spearman's rank correlation between each of the columns $1..m-1$ and column m and returns the maximum correlation, ρ_{max} and the expression $expr_{max}$ having the highest correlation. If the absolute value of ρ_{max} is greater than or equal to $\rho_{threshold}$, we have found the threshold expression, else we continue the search with expressions of larger size. Lines 9-15 extends the threshold expression depending on whether the correlation between the expression and work is positive or negative. The method GETSEQUENTIALTHRESHOLD returns the minimum/maximum value of $expr_{max}$ for which the call site must be executed asynchronously, depending on whether the work done by `f` increases or decreases as the value of $expr_{max}$ increases.

Algorithm 3 Threshold expression synthesis

Input: Profile data, P for call site `obj.f(`$\mathrm{a}_1..\mathrm{a}_n$`)`
Output: Threshold expression for call site `obj.f(`$\mathrm{a}_1..\mathrm{a}_n$`)`

```
1:  TExpr ← False
2:  for each T ∈ TYPES(obj) do
3:      if obj.f(a1..an) is conditional parallel for type T then
4:          for size = 1 to n do
5:              M ← CONSTRUCTMATRIX(P, size)
6:              (ρmax, exprmax) ← MAXSPEARMANCORR(M)
7:              if |ρmax| ≥ ρthreshold then
8:                  value ← GETSEQTHRESHOLD(M, exprmax)
9:                  if ρmax > 0.0 then
10:                     TExpr ← TExpr ∨ ((obj
11:                     instanceof T) ∧ (exprmax ≥ value))
12:                 else
13:                     TExpr ← TExpr ∨ (( obj
14:                     instanceof T) ∧ (exprmax ≤ value))
15:                 end if
16:                 break
17:             end if
18:         end for
19:     else if obj.f(a1..an) is parallel for type T then
20:         TExpr ← TExpr ∨ (obj instanceof T)
21:     end if
22: end for
```

Rule	*IFDS Results*	*Input Code*	*Output Code*
1	$\texttt{e} \notin \mathcal{M}(\texttt{s})$	`s: async e`	`async<T> { return e; }`
2	$\nexists \texttt{s}_1 : \texttt{x} \in \mathcal{N}(\texttt{s}_1)$	`T x`	`future<T> x`
3	$\exists \texttt{s}_1 : \texttt{x} \in \mathcal{M}(\texttt{s}_1) \wedge$ $\exists \texttt{s}_2 : \texttt{x} \in \mathcal{N}(\texttt{s}_2)$	`T x`	`mayfuture<T> x`
4	$\texttt{x} \in \overline{\mathcal{N}}(\texttt{s})$	`s: a = x.f;`	`T y = x.get();`
5	$\exists \texttt{s}_1 : \texttt{x} \in \mathcal{M}(\texttt{s}_1) \wedge$ $\exists \texttt{s}_2 : \texttt{x} \in \mathcal{N}(\texttt{s}_2) \wedge$ $\texttt{x} \notin \mathcal{M}(\texttt{s})$	`s: a = x.f;`	`T y = (T)x;` `a = y.f;`

6	$\exists \texttt{s}_1 : \texttt{x} \in \mathcal{M}(\texttt{s}_1) \wedge$ $\exists \texttt{s}_2 : \texttt{x} \in \mathcal{N}(\texttt{s}_2) \wedge$ $\texttt{x} \in \overline{\mathcal{N}}(\texttt{s})$	`s: a = x.f;`	`future<T> t = (future<T>)x;`
			`T y = t.get();`
			`a = y.f;`

7	$\exists \texttt{s}_1 : \texttt{x} \in \mathcal{M}(\texttt{s}_1) \wedge$ $\exists \texttt{s}_2 : \texttt{x} \in \mathcal{N}(\texttt{s}_2) \wedge$ $\texttt{x} \in \mathcal{M}(\texttt{s}) \wedge \texttt{x} \notin \overline{\mathcal{N}}(\texttt{s})$	`s: a = x.f;`	`T y;`
			`if (x instanceof future<T>)`
			`  future<T> t = (future<T>)x;`
			`  y = t.get();`
			`else`
			`  y = x;`
			`a = y.f;`

Table 6: Example transformation rules based on `future`-analysis results

References

[1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FM-CAD 2013*, pages 1–8, 2013.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95*, pages 207–216, New York, NY, USA, 1995. ACM.

[3] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP '12*, pages 3–8, New York, NY, USA, 2012. ACM.

[4] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[5] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT '01*, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[6] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ '11*, pages 51–61, New York, NY, USA, 2011. ACM.

[7] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI '12*, pages 89–98, New York, NY, USA, 2012. ACM.

[8] E. Coppa, C. Demetrescu, I. Finocchi, and R. Marotta. Estimating the empirical cost function of routines with dynamic workloads. In *CGO '14*, pages 230–239, New York, NY, USA, 2014. ACM.

[9] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *PLDI '98*, pages 106–117, New York, NY, USA, 1998. ACM.

[10] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *SC '08*, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[11] P. Feautrier and C. Lengauer. *Polyhedron Model*, pages 1581–1592. Springer US, Boston, MA, 2011.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[13] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95*, pages 209–220, New York, NY, USA, 1995. ACM.

[14] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *PLDI '11*, pages 458–469, New York, NY, USA, 2011. ACM.

[15] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7 (4):501–538, 1985.

[16] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP '07*, pages 251–264, New York, NY, USA, 2007. ACM.

[17] Y. He, C. E. Leiserson, and W. M. Leiserson. The cilkview scalability analyzer. In *SPAA '10*, pages 145–156, New York, NY, USA, 2010. ACM.

[18] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & Reiminfer: Checking and inference of reference immutability and method purity. In *OOPSLA '12*, pages 879–896, New York, NY, USA, 2012. ACM.

[19] S. Imam and V. Sarkar. Cooperative scheduling of parallel tasks with general synchronization patterns. In *ECOOP 2014*, volume 8586, pages 618–643. Springer Berlin Heidelberg, 2014.

[20] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09*, pages 3–14, New York, NY, USA, 2009. ACM.

[21] L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, Feb. 1974.

[22] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88*, pages 260–267, New York, NY, USA, 1988. ACM.

[23] J. Miettinen. Computer language benchmarks game. `http://benchmarksgame.alioth.debian.org/u64q/program.php?test=binarytrees&lang=java&id=6`. Accessed: 2015-11-06.

[24] A. Navabi and S. Jagannathan. Exceptionally safe futures. In *COORDINATION '09*, pages 47–65, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPoPP '08*, pages 23–32, New York, NY, USA, 2008. ACM.

[26] OpenMP. OpenMP specifications. `http://www.openmp.org/specs`.

[27] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for Java futures. In *OOPSLA '04*, pages 206–223, New York, NY, USA, 2004. ACM.

[28] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, New York, NY, USA, 1995. ACM.

[29] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The cilkprof scalability profiler. In *SPAA '15*, pages 89–100, New York, NY, USA, 2015. ACM.

[30] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15: 88–103, 1904.

[31] SPECjvm2008. Specjvm2008. `https://www.spec.org/jvm2008/`.

[32] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI'05*, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] J. Swaine, K. Tew, P. Dinda, R. B. Findler, and M. Flatt. Back to the futures: Incremental parallelization of existing sequential runtime systems. In *OOPSLA '10*, pages 583–597, New York, NY, USA, 2010. ACM.

[34] P. Thoman, H. Jordan, and T. Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *Euro-Par'13*, pages 164–177, Berlin, Heidelberg, 2013. Springer-Verlag.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99*. IBM Press, 1999.

[36] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA '05*, pages 439–453, New York, NY, USA, 2005. ACM.

[37] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI '12*, pages 67–76, New York, NY, USA, 2012. ACM.

[38] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in Java. In *PACT '07*, pages 130–139, Washington, DC, USA, 2007. IEEE Computer Society.

[39] L. Zhang, C. Krintz, and P. Nagpurkar. Supporting exception handling for futures in Java. In *PPPJ '07*, pages 175–184, New York, NY, USA, 2007. ACM.

[40] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09*, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.