The Eureka Programming Model for Speculative Task Parallelism

Shams Imam and Vivek Sarkar

Rice University, Houston, TX, USA, {shams,vsarkar}@rice.edu

Abstract

In this paper, we describe the Eureka Programming Model (EuPM) that simplifies the expression of speculative parallel tasks, and is especially well suited for parallel search and optimization applications. The focus of this work is to provide a clean semantics for, and efficiently support, such "eureka-style" computations (EuSCs) in general structured task parallel programming models. In EuSCs, a eureka event is a point in a program that announces that a result has been found. A eureka triggered by a speculative task can cause a group of related speculative tasks to become redundant, and enable them to be terminated at well-defined program points. Our approach provides a bound on the additional work done in redundant speculative tasks after such a eureka event occurs.

We identify various patterns that are supported by our eureka construct, which include search, optimization, convergence, and soft real-time deadlines. These different patterns of computations can also be safely combined or nested in the EuPM, along with regular task-parallel constructs, thereby enabling high degrees of composability and reusability. As demonstrated by our implementation, the EuPM can also be implemented efficiently. We use a cooperative runtime that uses delimited continuations to manage the termination of redundant tasks and their synchronization at join points. In contrast to current approaches, EuPM obviates the need for cumbersome manual refactoring by the programmer that may (for example) require the insertion of `if` checks and early `return` statements in every method in the call chain. Experimental results show that solutions using the EuPM simplify programmability, achieve performance comparable to hand-coded speculative task-based solutions and out-perform non-speculative task-based solutions.

1998 ACM Subject Classification D.1.3 [Programming Techniques] Concurrent Programming – Parallel Programming

Keywords and phrases Async-Finish Model, Delimited Continuations, Eureka Model, Parallel Programming, Speculative Parallelism, Task Cancellation, Task Termination

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.421

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.1.1.6`

1 Introduction

A wide range of problems, such as combinatorial optimization, constraint satisfaction, image matching, genetic sequence similarity, iterative optimization methods, can be reduced to tree or graph search problems [26, 1, 33]. A pattern common to such algorithms to solve these problems is a *eureka* event, a point in the program which announces that a result has been found. Such an event curtails computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. For example, in optimization problems, a eureka event declares (and updates) the currently best-known result and can prune the computation by causing the termination of specific tasks that cannot


© Shams Imam and Vivek Sarkar;
licensed under Creative Commons License CC-BY
29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 421–444

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


provide a better result. On the other hand, in satisfiability problems, the first eureka event can trigger the termination of the entire computation as a proof of existence has been found.

With the advent of the multicore era, future growth in application performance will primarily come from increased parallelism. While many efforts have focused on programming models that expose increased amounts of deterministic parallelism, we believe that it is also important to explore new programming model directions for speculative parallelism. Eureka-Style Computations (EuSCs) include search and optimization problems that could benefit greatly from speculative parallelism. However, writing parallel programs is a non-trivial, bug-prone, and complex endeavor in general, and the addition of speculation to current models can further add to the complexity. There is, hence, a need for programming models that support simple specification of parallel EuSC algorithms, combined with efficient parallel implementations.

One of the challenges in EuSCs is the efficient termination of multiple active tasks once a solution is published. Current termination techniques include the use of (a) terminating processes and threads [37], (b) exceptions for control flow as used in Microsoft's Task Parallel Library [25] and Java thread interrupts in blocking calls [34], (c) function-scoped cancellation points in Cilk `abort` [15] and OpenMP 4.0 [31], and (d) manual cooperative termination tokens as in Intel Thread Building Blocks [27]. As explained in Section 3, all these solutions have their limitations and are inadequate for supporting parallel EuSCs with high productivity and performance.

In this paper, we introduce the Eureka Programming Model (EuPM), an explicitly task parallel programming model that simplifies the expression of parallel EuSCs. The EuPM builds on a structured task parallel programming model (summarized in Section 2). It works by exploiting parallelism opportunities in computations that are divided into several *speculative tasks*; these tasks are called *speculative* because their results may or may not be needed. Section 3 motivates our approach to terminating speculative tasks once a result is published. Our solution uses a simplified cooperative termination technique that only requires a single method call at each eureka point - a point in a program that announces that a result has been found. The EuPM is described in Section 4. It promotes out-of-order executions and the constructs in our EuPM are expressive enough to encode many parallel programming patterns common to EuSCs (Section 5). These different patterns can also be safely combined or nested, thereby enabling both composability and reusability (Section 6).

We have implemented the EuPM as a Java-based task parallel cooperative runtime that runs on a standard Java Virtual Machine, and it is summarized in Section 7. Our approach could be implemented for parallel C++ programs as well. The burden of performing code transformations to ensure that all redundant tasks are terminated cleanly at well-defined program points is assumed by the compiler and runtime. We evaluate the performance of search and optimization benchmarks, when using standard task-based solutions, hand-coded cooperative speculative task-based solutions, and solutions based on our EuPM. Experimental results (Section 8) show that the EuPM solutions can deliver significant performance and productivity improvements over standard task-based solutions. The EuPM abstraction achieves acceptable overheads with performance that is comparable to that of hand-coded speculative task-based solutions, while the EuPM solutions are simpler to write. Section 9 discusses related work, and we summarize our conclusions and identify opportunities for future work in Section 10.

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

- Introduction of the Eureka Programming Model (EuPM) to simplify the expression and management of speculative parallel tasks, which are especially important for parallel search and optimization applications.

- A manifestation of the EuPM as a standard Java API, with compiler support for the cooperative termination of avoidable tasks at well-defined program points.
- Identification of various patterns that are well-suited for the eureka construct, but hard to implement using current parallel programming models. These patterns include search, optimization, convergence, and soft real-time deadlines.
- An implementation of the EuPM in a cooperative runtime for task parallelism that uses delimited continuations.
- An empirical evaluation of the productivity and performance benefits of the EuPM implementation on various EuSC benchmarks.

2 Background and Motivating Example

In the task-parallel model, the application execution can be modeled as a directed acyclic graph, where nodes represent computational tasks and edges define the data dependences among them. A runtime system then efficiently schedules tasks whose dependences have been satisfied over the available processing units, usually implemented as worker threads. The management of actual threads and related thread pools is done by the runtime and is transparent to the tasks in the program.

2.1 Async-Finish Programming Model

The Async/Finish Model (AFM) is a structured variant of the task-parallel Fork/Join Model. In the AFM, a task can *fork* a group of child tasks. These child tasks can recursively fork additional tasks. All these descendant tasks can potentially run in parallel with each other. Further, a parent/ancestor task can selectively *join* on a subset of child/descendant tasks to wait for their completion.

Tasks are created at *fork* points, the statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. An inner `async` is allowed to read and operate on a variable declared in an outer scope. The runtime is responsible for the scheduling of tasks created by `async`s. The `finish` construct represents a *join* operation. The task executing `finish` ⟨*stmt*⟩ has to wait for all transitively spawned child tasks inside ⟨*stmt*⟩ to terminate before it can proceed past the `finish` construct. All computations execute inside a global finish scope for the main program: the computation is allowed to terminate when all tasks nested inside the global finish terminate. This rule ensures that each executing task has a unique *Immediately Enclosing Finish* (IEF) [6, 40, 38].

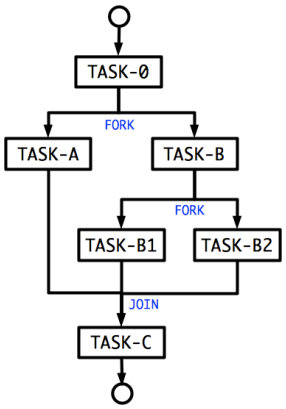
Listing 1 shows a sample program that uses `async` and `finish` constructs to preserve task dependences while exploiting available parallelism. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. The scopes of `async` and `finish` can span method boundaries that simplify parallelizing sequential programs. `async`s are inserted to wrap statements that can be executed in parallel and then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version. `async-finish` style computations are guaranteed to be deadlock-free [6]. In addition, in the absence of data races, these programs are deterministic [38].

2.2 Parallel Search of 2D Array

We will consider a well-understood parallel programming example as a motivating example where speculative computation may be used for parallelization. We build on this example in

Listing 1 `async-finish` program using `async` and `finish` constructs for synchronization. (Listings use pseudo-code syntax.)

```
1  class AsyncFinishPrimer {
2     def main(args) {
3        finish // the global finish
4           println("Task 0") // Task-0
5           finish
6              async // Task-A
7                 println("Task A")
8              async // Task-B
9                 println("Task B")
10             async // Task-B1 created by Task-B
11                println("Task B1")
12             async // Task-B2 created by Task-B
13                println("Task B2")
14          // wait for tasks A, B, B1 and B2
15          println("Task C") // Task-C
16    } }
```



the rest of the paper to discuss various concepts and explain different EuSC variants. The example performs the parallel search of a 2D array to find the index of a particular item if it exists. The eureka event occurs when the item is found. The parallel version may expand (or generate) more states than a serial version. We are ready to tolerate such redundancy in the hope of a faster execution time in finding the result.

Listing 2 displays an implementation of such a parallel search using the `async` and `finish` constructs. Using the `atomicRefFactory()` method, the program initializes a `token` container to invalid indices at line 6. Tasks are spawned at line 9 and enclosed in the finish scope declared at line 7. The finish scope ensures that all spawned tasks complete before the result is returned at line 14. To overcome the overheads of tasking, a common strategy while working with arrays is to chunk the data. Each `async` processes a chunk of data using the `for` loop at line 10. The eureka event occurs at line 19 when the search successfully finds a match to the goal and updates the `token` atomic variable. Since we are interested in any result, we use `set` instead of a `compare-and-swap` operation. After the eureka event, the `procRow` method promptly returns.

Listing 2 `async-finish` parallel search.

```
1  class ParallelSearch {
2    def atomicRefFactory() {
3      return new AtomicRef([-1, -1])
4    }
5    def doWork(matrix, goal) {
6      val token = atomicRefFactory()
7      finish
8        for rowIndices in matrix.chunks()
9          async
10           for (r in rowIndices)
11             procRow(matrix(r), r, goal, token)
12     // return either [-1, -1] or
13     // valid index [i, j] matching goal
14     return token.get()
15   }
16   def procRow(array, r, goal, token) {
17     for (c in array.length())
18       if goal.match(array(c)) // eureka!!!
19         token.set([r, c])
20         return
21   } }
```

Listing 2 highlights a few inefficiencies, also common to other EuSCs, while using `async-finish` style parallelism. Firstly, there is the need to pass the task-specific `token` variable to each relevant method in the call chain to allow triggering the eureka event. Secondly, returning early from `procRow` doesn't terminate the task as the task can continue to process other iterations of the `for` loop at line 10. Next, there are other tasks executing concurrently which need to be terminated. If these tasks are long running, there can be a potentially large wait time before the finish scope ends, and the result is returned. Finally, as per `async-finish` semantics, all tasks (including those sitting in the work queue) will be executed even

after the result is known. We discuss existing solutions to these problems in terminating tasks and the drawbacks of such solutions in Section 3.

3 Task Termination Strategies

It is well-known that speculative computation can yield performance improvements over conventional approaches to parallel computing [4, 32]. The speculative tasks can be started eagerly before they are known to be required, for example, by spawning parallel tasks to search disjoint fragments of a data structure. Once a solution is found, other attempts at solving the problem may be avoided (in optimization) or terminated (in search). Eagerly terminating such tasks improves performance by minimizing the amount of unnecessary computations. One of the challenges in EuSCs is the efficient coordination of the termination of several related tasks. Supporting termination requires ensuring that a task can stop gracefully and leave the system in a state that is known to be valid.

One approach is terminating processes and threads [37]. This is not a scalable solution as the runtime then needs to spawn additional processes or worker threads to maintain the parallelism in the application. When worker threads are terminated repeatedly, the overheads of resource initialization cause the performance of the application to degrade. In addition, terminating a task abruptly might cause a computation to be interrupted asynchronously which can cause havoc in the programmer's understanding of the code's behavior. As in [24], we believe that it should not be possible to terminate a task in any execution state, but only at places where certain program invariants hold such that the execution may be interrupted safely. As a result, a preemptive approach is not desirable, and a cooperative approach where the task actively decides when to terminate is preferred.

One cooperative approach is the use of exceptions for control flow [25, 34]. Using exceptions allows the task to terminate with the runtime providing the exception handler to process that exception. It has the benefit that only specific program points defined by the programmer need to be edited to insert the `throw` clause; the bodies of callees need not be modified[1]. Use of exceptions to terminate tasks fail when users provide custom handlers that inadvertently catch the exception being thrown. This prevents the exception from reaching the handler provided by the runtime and thus interferes with the termination logic. A compiler can rewrite the exception handlers to immediately rethrow these special exceptions and prevent user code from interfering with termination logic [2]. However, this policy fails to work in the presence of *inaccessible* functions (whose source code is not directly available for modification). In addition, native support for throw code is comparatively inefficient even in the absence of filling the stack trace. The frequent use of exception handlers for control flow program execution logic is expensive and should be avoided.

Another cooperative approach is function-scoped cancellation points [15, 31]. For example, possible locations for cancellation points in Listing 2 would be at lines 9 to 11 which include the scope at which the `async` was declared. These work better as the compiler rewrites the code to support task termination; however, the limitation is that cancellation is not possible when the code is executing in a nested function call.

Another approach is manual cooperative termination via cancellation tokens or interrupt checking [27, 34]. Within long-running tasks, manually inserted periodic termination checks allow the task to determine if further work is avoidable (i.e. the task can be terminated). The granularity of checks controls the trade-off between the responsiveness of termination of

[1] Callee signatures may need to be modified to include the exception, however.

Listing 3 Parallel search with manual cooperative termination.

```
1  class ParallelCooperativeSearch {
2     def atomicRefTokenFactory() {
3        return new AtomicRefToken([-1, -1])
4     }
5     def doWork(matrix, goal) {
6        val token = atomicRefTokenFactory()
7        finish
8           for rowIndices in matrix.chunks()
9              async
10                for r in rowIndices
11                   procRow(matrix(r), r, goal, token)
12                   // cooperative termination check
13                   if token.isResolved()
14                      return
15       return token.get()
16    }
17    def procRow(array, r, goal, token) {
18       for c in array.length()
19          // cooperative termination check
20          if token.isResolved()
21             return
22          if goal.match(array(c))
23             token.set([r, c])
24             return
25    } }
```

tasks and the overhead of such check calls. Listing 3 displays the program from Listing 2 with support for manual cooperative termination. This approach is cumbersome to write as the programmer needs to manually transform all methods to support this style with an additional `token` parameter. It also requires `if` checks and early return statements (lines 13-14 and 20-21). Inserting such checks in the source code is awkward and impossible in the case of calls to inaccessible functions. If the computation includes inaccessible functions in the call stack, we need to wait for the body of each such function to complete before termination can be effected.

3.1 Delimited Continuation-based Cooperative Termination

Delimited Continuations (DeConts) were introduced by Felleisen in 1988 [14] where he referred to them as *prompts*. Other variants for DeConts include the `shift/reset` mechanism introduced by Danvy and Filinski [10]. Continuations represent the rest of a computation from any given point. They refer to the ability to *capture* the state of a computation at that point; the computation can later be *resumed* from that point by resuming the continuation. In contrast, DeConts represent the rest of the computation from a well-defined outer boundary, i.e. a sub-computation. DeConts work even in the presence of inaccessible functions in the call stack. When a computation is suspended, DeCont cause control to return to the caller of the boundary function irrespective of the functions (including inaccessible ones) that are in the call path. DeConts are hence a good choice when a limited part of the computation needs to be saved/restored [11]. Many mainstream languages offer support for various forms of DeConts, such as Boost coroutines in C++ [29], Kilim framework in Java [41], `shift/reset` in Scala [39], Ruby fibers [21], etc. DeConts are notorious for being hard to use and to understand by developers (as opposed to compilers and runtime systems). Hence, in a system that uses DeConts, an approach that does not expose a developer to DeConts is desirable.

Our approach to termination is cooperative and relies on the transparent use of DeConts. DeConts are required to let the control return to the runtime by safely unrolling the task's call stack but not the runtime worker's call stack, after which the runtime can perform

cleanup as if the task terminated or returned normally. The termination approach involves coordination between the code that requests termination (task that resolves a eureka) and the task that responds to termination (other executing tasks that have become redundant). Our approach guarantees that early termination of tasks can only occur at at well-defined program points: one of the check points or at points that resolve a eureka. If a task needs to be terminated, the call stack is unwound and control returned to the runtime via the use of DeConts. Such tasks are not resumed, so, unlike general DeConts, the state of the computation at that point need not be saved. Introducing a method call for the check introduces some overhead, so tasks can decide if they are *terminable* and how responsive they want to be to interruption. We expect a sweet spot, where a balance is reached between the overheads incurred by checks and the gains from early task termination. The optimal frequency of checking is application-specific and is determined by the developer. While the programmer does not need to hard-code exceptions and `if` checks on cancellation, our approach attempts to merge the benefits of using:

(a) **Exceptions:** There is no need for repeated `if` checks every time a method returns from a terminable method in the call chain. Only specific program points need to be edited to insert a method call. No exceptions are used for control flow, and the approach is unaffected by the presence of exception handlers.

(b) **Cancellation tokens:** It allows the programmer to determine the frequency of checks to terminate a task and determine responsiveness. However, our approach does not require changing the signature to add an additional parameter, the cancellation token is discovered implicitly by the task (further described in Section 4). Also, the body of inaccessible functions do not need to complete before termination of a terminable task can be effected.

The main limitation of our cooperative approach is that the programmer has to determine the frequency of the check calls and manually insert the termination check calls. These termination checks can also be inserted automatically by a compiler. Previous work by Feeley on *balanced polling* [13] and in the Jikes VM *yieldpoints* [44] provide a scheme to automatically insert these calls while minimizing overhead. Another limitation of our approach is that termination should not be triggered in a critical section that is implemented with programmer-defined locks. Acquiring a lock but failing to ensure that it is released can cause termination of the overall computation to be arbitrarily delayed; this issue plagues all the cooperative schemes discussed above. The issue can be circumvented by the use of managed resource control techniques that allow the runtime to track the lock(s) obtained by the user code while executing critical sections. These locks obtained by a task can then be released by the runtime when requested to do so. Such techniques include the use of custom virtual machines or language constructs to execute the critical section. For example, the Habanero programming model [23] includes such a language construct, called `isolated`, which would work with terminated tasks.

4 Programming with `Eurekas`

Parallel programming models ideally enable programmers to express parallel algorithms using abstractions that hide all but the relevant information to reduce complexity and to increase programmer productivity. Our goal is to define the Eureka Programming Model (EuPM) so that it can be used to write parallel programs for EuSCs more productively than current parallel programming models. In this section, we first introduce the `Eureka` construct that is used by speculative tasks to trigger eureka events. Next, we describe how parallelism is

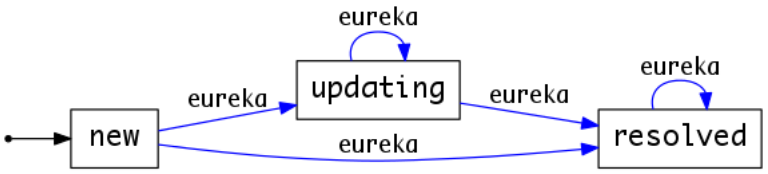


Figure 1 Life-cycle of `Eureka`. The states and transitions will become clearer when we introduce the different `Eureka` patterns in Section 5.

expressed via speculative tasks in the EuPM. We also explain how the termination of a single task, as well as a group of tasks, is supported in the EuPM.

4.1 `Eureka` Construct and API

A `Eureka` is a new construct that provides support for speculative parallelism in an `async-finish` setting. Once a `Eureka` construct has been *resolved* by reaching a stable value, it enables detection of a group of speculative tasks that can be terminated. It abstracts away implementation details, facilitates encapsulating any mutable state, and provides an API to allow tasks to concurrently notify eureka events as well as to query the state of the eureka object. Encapsulation simplifies data race avoidance while attending to concurrent eureka events triggered by speculative tasks.

As seen in Figure 1, a `Eureka` object has a well-defined life cycle; it can only transition between the states in response to eureka events. During its life cycle, a `Eureka` is in one of the following states:

(a) **new**: an instance of the `Eureka` has been created and initialized; however, it has not yet received any eureka events.

(b) **updating**: the `Eureka` has received at least one eureka event, and its internal state has not reached a stable value (e.g. computing minimum during optimization).

(c) **resolved**: the `Eureka` has reached a stable value; any subsequent eureka events may be ignored. Once a `Eureka` enters the resolved state, all speculative tasks that can trigger eureka events to update this `Eureka` become terminable and can be terminated.

We have developed an interface which supports basic behavior needed by tasks in EuSCs. This `Eureka` interface can be used to support a wide variety of patterns in EuSCs including search, optimization, and convergence. User's can also use this interface to build their custom implementations for `Eureka`s. The operations that can be performed on a `Eureka`, `eu`, are defined by the following interface:

(a) `offer(auxiliaryData)`: Notifies `eu` that a eureka event has been triggered; additional information used to mutate the internal state of `eu` is available in `auxiliaryData`. This operation enables `eu` to transition to different states in its lifecycle (as shown in Figure 1). Whether or not the event resolves `eu`, it can cause the task invoking this operation to terminate at a well-defined program point.

(b) `check(auxiliaryData)`: This operation allows a speculative task to check whether it has become terminable as, e.g., `eu` has been resolved. If the task has become terminable, a call to `check` will cause the task to be terminated. By accepting an argument, `check` enables the caller to pass additional values that can be used to determine whether to terminate a task.

(c) `isResolved()`: Allows a speculative task to query whether `eu` has been resolved. This method returns a boolean value and never causes a task to be terminated.

(d) `get`(): If `eu` has been resolved, it returns the resolved value. Otherwise, a transitory value of `eu` is returned. One is guaranteed to receive the resolved value if this operation is invoked outside the `finish` scope on which `eu` was registered (e.g. as we will see in the explanation of Listing 4 line-12).

4.2 Eureka Programming Model (EuPM)

The EuPM is an extension of the task-parallel `async-finish` model where speculative tasks are created using the `async` keyword. Through the hierarchical nature of `async` and `finish` blocks, we advocate a structured approach to parallel programming of EuSCs. The task executing a `finish` has to wait for all transitively spawned child tasks created inside the finish scope to terminate before it can proceed. A `finish` block can register on a `Eureka`, `eu`, with the following pseudocode syntax (the library API includes `eu` as a parameter to the `finish` API): `finish`(eu) ⟨*stmt*⟩. The `finish` construct simplifies the identification of the group of tasks that participate in a eureka-style synchronization on a particular `Eureka` instance.

All tasks having the same immediately enclosing finish (IEF) belong to the same group and inherit the registration on the `Eureka` instance, `eu`, from the finish scope. Finish scopes with different `Eureka` instance registrations can be nested allowing composability of different speculative computations. Similarly, multiple `finish` blocks can register on the same `Eureka` instance, `eu`, to represent that different speculative sub-computations are linked. When one of the speculative tasks resolves `eu` it makes other tasks from the same or different groups also registered on `eu` to become redundant and terminable. If none of the tasks trigger a eureka event that resolves the registered `eu`, the computation completes normally when all tasks inside each `finish` scope complete.

The EuPM specific operations that a task, `T`, can perform on a `Eureka`, `eu`, are defined as follows:

(a) **new**: Task `T` can create a new instance of the `Eureka` construct, `eu`, and obtain a handle to it. The reference `eu` can now be used to register on new `finish` scopes. The creator task can pass the reference of `eu` to other tasks.

(b) **registration**: `eu` can be explicitly registered on a `finish` scope. Note that the task that created `eu` cannot register on `eu`. A newly spawned task, `T`, implicitly registers on `eu` only if the IEF of `T` was explicitly registered on `eu`. Currently, we do not provide a mechanism for an `async` task to explicitly register on `eu`.

(c) **offer**: This method retrieves the `Eureka` instance, `eu`, that the task is registered on and invokes the `eu.offer`() method to notify `eu` of a eureka event. This simplifies the computation body of `T` where method calls do not need to add an extra parameter to pass `eu` down the call chain. Invoking this method can cause the task to terminate (depending on the implementation of `eu`).

(d) **check**: A task indirectly performs a check on `eu` by invoking the static `check` method. Like the previous operation, it retrieves `eu` to make the call `eu.check`(). Invoking this method can cause the task to terminate (depending on the terminating logic of `eu`), so programmers need to ensure that side-effects introduced by `T` are in a consistent state at the program point where this method was invoked.

With the EuPM, a programmer can focus on writing operational code explicitly specifying potentially parallel operations, leaving the underlying details of parallel execution and termination detection to the runtime system. The EuPM places no requirements on the use of a shared-memory infrastructure. Like the `async-finish` model, the EuPM presented in this paper is also applicable to a distributed environment. One of the key features of a

system that supports EuSCs is the efficiency with which the eureka events are triggered. The EuPM provides the abstraction, static `offer` method, that simplifies how eureka events are triggered in tasks. Invalid calls to `check`/`offer` from a task not executing in a EuSC (i.e. `finish` scope not registered on a `Eureka`) results in a runtime error.

Another feature is the efficiency with which the state of the program can be updated after the result has been found. The EuPM needs to provide a means to easily detect tasks that need to be terminated and a mechanism to guarantee effective termination of terminable tasks. Once a `Eureka` instance, `eu`, moves to the resolved state, all incomplete tasks belonging to a finish scope registered on `eu` become terminable. Any ready tasks belonging to the *resolved* finish scope can be terminated by removing them from the work queue – each task is assumed to contain an implicit `check` call at the start of the task execution. Redundant executing tasks are terminated at program points where the `check` method is invoked. This allows tasks to terminate cooperatively in a programmer controlled manner and, more importantly, simplifies reasoning about the correctness of the speculatively parallel program.

Listing 4 displays the parallel search program of Listing 2 using `async` and `finish` constructs in the EuPM. We create the `SearchEureka` instance, `eu`, inside the factory method `eurekaFactory`. This instance, `eu`, is registered by the `finish` scope defined on line 7. Hence, all `async` tasks launched at line 9 are automatically registered on `eu` and belong to the same group. The tasks trigger the eureka event by invoking the `offer` method at line 18. There is no need for an explicit `return` statement in `procRow`, as `offer` on a `SearchEureka` causes the task to terminate. To enable cooperative termination, there are also calls to `check` (line 16) to check the state of the registered eureka implicitly. When `eu` has been resolved, `check` will cause the terminable tasks to terminate. Eventually, all tasks inside the `finish` block at line 7 will complete execution or be terminated, and the computation will proceed to line 12 and the result will be returned. Note that, like Listing 3, the final answer in this example is nondeterministic, but there are no data races involved. It should be noted that this program (19 lines) required fewer lines of code than the equivalent program in Listing 3 (25 lines). In addition, the code in Listing 3 is more complicated and error-prone than the code in Listing 4. As we will see in Section 5, we will build on this example to explore various EuSC patterns with minimal change in the kernel code (lines 5 to 19 of Listing 4).

Listing 4 Parallel search using the Eureka model.

```
1  class ParallelEurekaSearch {
2     def eurekaFactory() {
3        return new SearchEureka([-1, -1])
4     }
5     def doWork(matrix, goal) {
6        val eu = eurekaFactory()
7        finish (eu) // eureka registration
8           for rowIndices in matrix.chunks()
9              async
10                for r in rowIndices
11                   procRow(matrix(r), r, goal)
12       return eu.get()
13    }
14    def procRow(array, r, goal) {
15       for c in array.length()
16          check([r, c]) // coop. term. check
17          if goal.match(array(c))
18             offer([r, c]) // trigger eureka
19    } }
```

5 Parallel Patterns and Eureka Variants

In this section, we describe frequently occurring patterns that arise in EuSCs and how they can be solved using the EuPM. These patterns include computations that produce both deterministic and non-deterministic results.

5.1 Parallel Search

Search is a well-known pattern in EuSCs. It is a non-deterministic computation in the sense that if the goal is present at multiple locations in the data structure being searched, any of those locations is an acceptable result. Searching disjoint partitions of a data structure can be done in parallel though it may considerably increase the amount of work that the algorithm performs. Such parallelism is speculative since more than one partition may contain a solution. Once the result is discovered, all parallel searching entities should ideally be terminated as quickly as possible to minimize doing redundant work. With respect to the EuPM, this means that the first eureka event triggered by a task will resolve the `Eureka` instance, `eu`, registered by the task. Hence, a `SearchEureka` construct is designed to be resolved by the first eureka event it processes, and it promptly terminates the task that triggered the event. Any subsequent calls to `check` or `offer` by other tasks registered on `eu` result in those tasks being terminated.

The same concept can be used for termination detection in the `finish` statement with regards to exception semantics. If any `async` throws an exception, then it can resolve an implicit `SearchEureka` registered by the `finish` scope. All other `async`s belonging to the same IEF can then be terminated at their next `check`/`offer` checkpoint. The IEF can then rethrow the exception thrown by the `async`. This offers an alternate strategy to the *MultiException* scheme [6] where a collection of all exceptions thrown by all `async`'s in the IEF is rethrown.

5.2 Count Eureka

Another variant of a parallel search is where we wish to know the first K results that match a query. This pattern is inspired by the `ParallelTry` command in Mathematica 7 [47]. In this pattern, we wish to terminate the computation when at least K of the asynchronous computations have completed successfully. Any evaluations still underway after K results have been received are avoidable and should be terminated. Like the `SearchEureka` pattern, the `CountEureka` pattern produces non-deterministic results as the results received are dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events.

The program from Listing 4 can be modified to use the `CountEureka` construct by changing the factory method. A `CountEureka` is initialized with a count K and is resolved after exactly K eureka events have been triggered. Once resolved, any calls to `offer` and `check` cause the calling task to be terminated. A call to `CountEureka.get()` returns a list of values of maximum length K instead of a single value. If none of the tasks triggered a eureka, then an empty list is returned. In general, a `SearchEureka` can be viewed as a `CountEureka` with a count of 1.

5.3 N-Version Eureka

N-Version Programming [7] uses software redundancy to achieve fault-tolerance. In N-Version Programming, there are multiple functionally equivalent implementations of the same specification. These implementations can run be independently in parallel to compute results; the results are notified using eureka events. Using a decision algorithm, such as when any N (≥ 2) agree on their results, the eureka is resolved. The agreed upon value is accepted as the result matching the specification, and other computations are terminated. This pattern also produces non-deterministic results as the final result is dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events from the independent implementations.

5.4 Optimization Eureka

Many problems from artificial intelligence can be defined as combinatorial optimization problems. For example, Branch and Bound (BnB) is a widely used tool for solving large-scale NP-hard combinatorial optimization problems [8]. A BnB algorithm searches the complete space of solutions for a given problem for the best solution. Subproblems are derived from the originally given problem through the addition of new constraints. An objective function computes the lower/upper bounds for each subproblem. The upper bound is the worst value for the potential optimal solution; the lower bound is the best value. The entire tree maintains a global upper bound (GUB): this is the best upper bound of all nodes. Nodes with a lower bound higher than the GUB are eliminated from the tree because branching these sub-problems will not lead to the optimal solution. In many practical cases, the amount of pruning that occurs in this type of BnB algorithm can be very significant.

In parallel implementations, pruning the branches of the search tree may lead to terminating existing computations. The structure of the BnB search requires the ability to terminate individual subtrees of the search tree [35]. A BnB version of our array search example is where we are interested in finding the lowest index of the goal item if it exists in the array. We can achieve this by modifying the factory method in Listing 4 to return a `MinimaEureka` instance. In our EuPM, the GUB is available in the `MinimaEureka` instance, `eu`, that a speculative task is registered on and can be retrieved by a call to `eu.get()`. Calls to `check` and `offer` pass the current known upper bound or solution, respectively, as the argument. If the argument in the `offer` call is lower than the GUB, the GUB is updated in the `MinimaEureka` instance, otherwise the current task is terminated. Similarly, calls to `check` terminate a task if the argument is larger than the currently known GUB in `eu`.

5.5 Soft Deadlines

For soft real-time systems [5] the goal is to meet a certain subset of deadlines to optimize some application-specific criteria. If a soft real-time task takes longer than the allotted time since its creation to complete, then it needs to be terminated with its latest results. Another similar notion is that of engines that abstract the notion of timed preemption [20]. An engine is run by giving it a quantity of abstract time units that measure computation. If the engine completes its computation before running out of units, it returns the result of its computation and the quantity of remaining units. If it runs out of units, the computation is terminated. Unlike Haynes' original notion of engines, nesting of engines is allowed in our model thus allowing time units to be divided among parallel sub-tasks if required.

In our soft deadline version of the array search example, the deadlines could be overall execution time (soft real-time) or number of comparison operations performed (abstract time units). The eureka version of these programs helps the system by releasing the resources of the tasks which have already missed their deadlines and allocating more resources to the other tasks which can still potentially meet their deadlines. We support both kinds of `Eureka`s in the form of `TimerEureka` and `EngineEureka`, respectively. A `TimerEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time since the creation of `eu`. An `EngineEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time units (measured by the sum total of the arguments to `check`). These `Eureka` instances can trigger the termination of a group of tasks without an explicit `offer` from a task. Note that since tasks only get cancelled when they invoke `check`, tasks can run for much longer than the allotted time unless the user is careful with the calls to `check`. This is consistent with our philosophy of allowing the programmer to determine responsiveness (Section 3).

5.6 Convergence Iterations

Iterative methods [17] refer to a wide range of techniques that use successive approximations to obtain more accurate solutions to a set of equations at each step. Examples of iterative methods include the Jacobi method, Gauss-Seidel method, and the Successive over-relaxation method. An iterative method is called *convergent* if the corresponding sequence converges for given initial approximations. Speculatively parallelizing an iterative algorithm results in creating tasks for computations of *future* iterations.

Listing 5 Example of an iterative method using the Eureka model.

```
1  class ParallelEurekaConvergence {
2    def eurekaFactory(initVal, tolerance) {
3      val pred = (a, b) -> {
4        Math.abs(decimalDiff(a, b)) <= tolerance
5      }
6      return new ConvergenceEureka(initVal, pred)
7    }
8    def doWork() {
9      val maxIters = 100
10     val initVal = INFINITY
11     val tolerance = 1e-4
12     val eu = eurekaFactory(initVal, tolerance)
13     finish (eu) {
14       // arrays to store task handles
15       val xs = newArray(maxIters + 1)
16       val ys = newArray(maxIters + 1)
17       val rs = newArray(maxIters + 1)
18       xs[0] = async { return xInit }
19       ys[0] = async { return yInit }
20       rs[0] = async { return initVal }

22       for (i in 1 to maxIters) {
23         xs[i] = async {
24           await(xs[i - 1]) // dependence
25           check()
26           return f(xs[i - 1])
27         }
28         ys[i] = async {
29           await(ys[i - 1]) // dependence
30           check()
31           return g(ys[i - 1])
32         }
33         rs[i] = async {
34           check()
35           await(xs[i], ys[i], rs[i - 1])
36           val iterRes = h(xs[i], ys[i])
37           // converge if rs[i] and rs[i-1] close
38           offer(iterRes)
39           return iterRes
40     } } }
41     return eu.get()
42 } }
```

Listing 5 displays an example which computes r using the equation: $x_{i+1} = f(x_i)$, $y_{i+1} = g(y_i)$, $r_i = h(x_i, y_i)$ and converges when successive values of r become *close*. The parallel version launches `maxIters` iterations ahead of time (line 22) and parallelizes the computation to respect the dependences (using the `await` clause). The `await(T1, ..., TN)` clause in a task causes it to be suspended from execution until the execution of tasks `T1, ..., TN` has completed. The `await` clause in line 35 ensures that values of r are offered to `eu` in the expected order. Once convergence is reached (i.e. when true is returned by the call to `predicate` inside `eu` with the current and new value) `eu` is resolved and all tasks spawned by the computation need to be terminated. Note that this includes tasks that may be transitively suspended on `await` clauses as each await task is assumed to contain an implicit `check` call at the start of task execution (Section 4.2). Calls to `check` in the `async`s and possibly inside methods `f`, `g`, and `h` ensure executing tasks can be terminated early.

6 Reusability and Composability of Eureka Components

Abstractions and productivity are among the most important requirements for programming models. Further, it is important for the abstractions to be as orthogonal as possible, so as to aid composability and reusability. Constructing reusable components aids in programmer productivity by simplifying building of larger systems from relatively simpler parts. While current approaches to support EuSCs lack composability and reusability in general, we show in this section that all the different styles of eureka computations presented in Section 5 can be safely combined or nested thereby enabling general composability and reusability.

Listing 6 Example of a parallel search on two elements of an 2-D array using the Eureka model.

```
1  class ParallelEurekaDoubleSearch {
2     def eurekaFactory() {
3        val initValue = [-1, -1]
4        return new SearchEureka(initValue)
5     }
6     def doWork(matrix, goal1, goal2) {
7        val eu1 = eurekaFactory()
8        val eu2 = eurekaFactory()
9        val eu = eurekaComposition(AND, eu1, eu2)
10       finish (eu) // eureka registration
11          for rowIndices in matrix.chunks()
12             async
13                for r in rowIndices
14                   procRow(matrix(r), r, goal1, goal2)
15       // eu.get() returns pair of values
16       return eu.get()
17    }
18    def procRow(array, r, g1, g2) {
19       for c in array.length()
20          // pair of values for eu1 and eu2
21          val checkArg = [[r, c], [r, c]]
22          // cooperative termination check
23          check(checkArg)
24          val loopElem = array(c)
25          val res1 = g1.match(loopElem) ? [r, c] : null
26          val res2 = g2.match(loopElem) ? [r, c] : null
27          // pair of values for eu1 and eu2
28          val foundIdx = [res1, res2]
29          // possible eureka event
30          offer(foundIdx)
31    } }
```

6.1 Composability by Component Composition

Component composition involves the systematic combining of independent components to form useful components. Such incremental aggregation of existing components yields further components. This method is scalable as code replication is avoided while implementing new functionality. In the EuPM, independent `Eureka`s can be combined to form new `Eureka`s. The constituent `Eureka`s are used as encapsulated black-box components and are accessed solely through their exposed interfaces (the `check` and `offer` operations).

We support basic logical conjunctive and disjunctive binary composition semantics for `Eureka`s. Calls to `offer` and `check` need to be passed a pair of values, one for each component `Eureka`. The conjunctive composition of `Eureka`s is considered resolved only when both the constituent `Eureka`s are resolved. A disjunctive composition of `Eureka`s is considered resolved when either of the constituent `Eureka`s is resolved. Since the state of a `Eureka` instance evolves monotonically (once resolved a `Eureka` always remains resolved), the binary composed `Eureka` also preserves monotonicity, i.e., resolution of a `Eureka` is a stable property.

Listing 6 shows an example with a conjunctive eureka. We extend the example from Listing 4 to search for two target items in parallel and report a success only when both items are found. The example creates two search `Eureka`s and then creates a conjunctive eureka, `eu`, at line 9. This `eu` is used to register on the finish scope and launch the parallel search tasks on individual rows. Each call to `check` and `offer` now passes a pair of values (lines 23 and 30). The internal implementation delegates the individual values from the pairs to the component `Eureka`s. These individual values may end up resolving only one of the component `Eureka`s. The overall conjunctive eureka, `eu`, is resolved when both component `Eureka`s are resolved, possibly from different calls to `offer` in different tasks. Once `eu` has been resolved, calls to `check` (line 23) will result in the task being terminated.

6.2 Reusability by leveraging Functional Decomposition

Function decomposition refers to the process of splitting a computation into multiple fragments. These fragments are invoked as helper functions and their results combined to produce the overall result. Alternatively, it can also be viewed as the functional composition of existing functions so that the results of these function calls are used to evaluate a larger computation. Reuse is achieved by building functions on top of existing functions. Such uses of functions for individual computation fragments simplify maintainability and avoid code duplication.

In the EuPM, we enable opportunities for functional decomposition by allowing the nesting of EuSCs with `finish` and `async` statements. This nesting can be arbitrarily deep and allows exposing nested fork-join parallelism opportunities in distinct EuSCs. Nesting of `finish` blocks registered on `Eureka` instances is allowed and enables composability of different speculative computations. Each `finish` scope registered on the same `Eureka` instance forms a single group. When different EuSCs are nested, calls to `offer` are delegated to the registered `Eureka` on the IEF of the currently executing task. However, calls to `check` are recursively delegated to registered `Eureka` instances up the nested `finish` scope hierarchy. This allows the innermost EuSC to continue to work as before, but tasks may be aborted if `Eureka`s up the hierarchy have been resolved by other parallel tasks. Thus, nesting EuSCs causes a tree hierarchy of `Eureka`s to become linked whereby resolving a `Eureka` up the hierarchy causes computations lower in the tree to be terminated.

This nesting mechanism is explained in Listing 7 which shows an example to do a search in a multidimensional array. The solution reuses the `doWork` function from Listing 4 and is thus performing functional composition. Nested EuSCs are also created at line 14 where an individual `Eureka` instance is created for every row (line 13) in the leading dimension of the array in the recursive call. When the `Eureka`, `eu`, is resolved for a dimension `N1`, it causes all nested eureka computations processing dimension `N2` (where `N2` $<$ `N1`) to be treated as resolved. As a result, subsequent calls to `check` (at line 17) in the redundant tasks of the nested computations will cause the tasks to terminate. After returning from a call to a nested eureka computation, the result is either returned immediately (for the base case at line 11) or processed further (line 18). Further processing involves updating the result index with the value for the current dimension before offering the result via `offer` (line 20).

7 Implementation

Despite any productivity promises, a parallel programming model must be implementable in an efficient and scalable fashion for it to be accepted by programmers. Our implementation of the EuPM is an extension of a Java-based task parallel runtime [23] that supports cooperative scheduling of `async-finish` style computations, though our ideas can also be implemented in other languages including C/C++. Figure 2 highlights the features and responsibilities of the task-parallel Eureka runtime in our model. These include implementation of efficient eureka variants; management and scheduling of the speculative tasks; classification of tasks into eureka groups; termination of redundant tasks; and synchronization and coordination of tasks

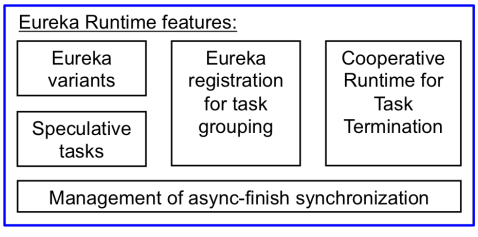


Figure 2 Features supported by a Eureka task parallel runtime.

Listing 7 Example of parallel search on a multidimensional array using function decomposition and nested eureka computations in the Eureka model.

```
class ParallelEurekaArraySearch {
   def eurekaFactory(dim) {
      val initValue = array(dim).fill(-1)
      return new SearchEureka(initValue)
   }
   def doWork(matrix, dim, goal) {
      if (dim < 2)
         throw IllegalArgumentException("Invalid dimension: " + dim)
      else if dim == 2
         // reuse by call to existing eureka computation  Listing 4
         return ParallelEurekaSearch.doWork(matrix, goal)
      else
         val eu = eurekaFactory(dim)
         finish (eu) // nested eureka registration from recursive calls
            for i in matrix.length()
               async
                  check(array(dim-1).fill(-1).insert(i)) // termination check
                  val resIndices = doWork(matrix(i), dim-1, goal)
                  if isIndexNonNegative(resIndices)
                     offer(copy(resIndices).insert(i))
         return eu.get()
} }
```

inside `finish` blocks. The challenges in the implementation of the EuPM involve effective load balancing of tasks, terminating tasks in a group efficiently, and supporting `Eureka`s in a scalable manner.

Our implementation supports all the `Eureka` variants described in Section 5 based on the API defined in Section 4.1. The key challenge is to support the synchronization by keeping each thread busy without any blocking. We rely on Java's support for atomic variables to implement the `Eureka`s and detect when a `Eureka` has been resolved. In particular, we rely heavily on the use of *compare-and-swap* operations, on `AtomicBoolean` and `AtomicLong` instances. This ensures thread safety and avoids data races in the concurrent calls to `check` and `offer` on the `Eureka`.

Our runtime uses the help-first policy [18] and maintains an independent stack for each task. Hence, any worker thread may execute a task, and we can use either work-stealing or work-sharing scheduling policies in our runtime. Since subproblems are generated and consumed dynamically, we rely on the load balancing algorithm provided by Java's `ForkJoinPool`. The `ForkJoinPool` is an optimized thread pool, which employs a work-stealing mechanism to efficiently distribute submitted tasks among its pool threads. Figure 3 displays how the runtime uses work-stealing threads to schedule tasks. Each task also maintains a reference to its IEF and inherits the `Eureka` registration from its IEF. This `Eureka` instance is stored in the IEF and used as the recipient while delegating calls to `check` and `offer`. Thus the tasks registered with the same `Eureka` instance are implicitly grouped together and become terminable when the `Eureka` is resolved. The tasks are eagerly terminated when there is a call to `check` or `offer`. Tasks executing inside a `finish` scope not registered with a `Eureka` (e.g. `FIN-0`) execute as regular `async-finish` style tasks without support for early termination.

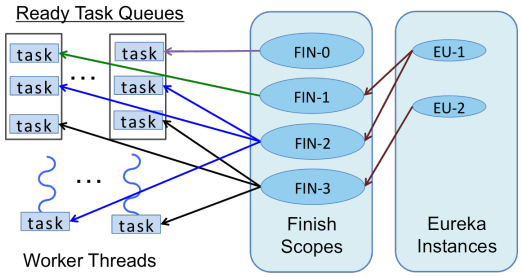


Figure 3 Execution of parallel Eureka tasks in a work-stealing environment.

As mentioned in Section 3, we rely on a cooperative task termination technique. When a task needs to be terminated at one of the check points, the call stack is unwound and control returned to the runtime via the use of Delimited Continuations (DeConts) [14]. Our implementation conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for DeConts or for storing and restoring the stack that we need to support cooperative termination. We have built a cooperative runtime that schedules tasks in the presence of end-of-finish synchronization constraints without blocking underlying worker threads using the strategy described in [22]. The DeConts created are thread independent and can be resumed on any worker thread. This strategy is known to provide a more scalable solution than other schemes that use thread-blocking operations [22].

We use an extended version of the open source bytecode weaver provided by the Kilim framework [41] to support DeConts. The Kilim bytecode weaver works by transforming the code of methods which can terminate. It recognizes such methods by the presence of a `SuspendableException` exception in the method signature. It is important to note that no actual exceptions are thrown or caught which minimizes the overhead of capturing and resuming continuations. Instead, the transformation performed is similar to a continuation passing style transformation, except that only methods that can suspend are transformed. The weaver also allocates custom state objects to store local variables to enable restoring the computation. We extended Kilim to enable execution of parallel tasks run by worker threads in the `ForkJoinPool`. Then we added support for terminated tasks. Such tasks are never resumed, so unlike general DeConts the state of the computation at that point need not be stored. This avoids additional memory allocation and garbage collection overheads while suspending the DeCont.

Next, we extended the classical `async-finish` constructs with support for the proposed `finish` and `eureka` constructs used in EuSCs. We provide support for EuPM `finish` constructs that register on a `Eureka`, `eu`, and each time a task is spawned from the finish scope, the task is also registered on `eu`. Nested finish scopes can register on different `Eureka` instances as each finish scope maintains its own `Eureka`. This enables composability of different speculative computations. Static helper methods, such as `check` and `offer`, then retrieve the `Eureka`, `eu`, registered with the currently executing task from its IEF before delegating the call on `eu`. Whenever a eureka is resolved, the finish scope, `f`, is notified, and all tasks whose IEF is `f` and that are in the work queue are terminated immediately. In-flight executing tasks belonging to `f` are terminated at `check` or `offer` points as termination is cooperative. Terminating executing tasks is done by suspending the current DeCont and flagging it as terminated, so that the runtime can perform cleanup operations and schedule other tasks for execution. Once all of these tasks have been successfully terminated, the end-of-finish point for `f` is resumed.

8 Experimental Results

The benchmarks were run on individual nodes in an IBM POWER7 compute cluster. Each node contains 256GB of RAM and four eight-core IBM POWER7 processors running at 3.8GHz each. The software stack includes IBM Java SDK Version 7, and our implementation of the cooperative runtime version 0.1.2. We configured each benchmark to run using 32 worker threads and run for thirty iterations in six separate JVM invocations using the same

Table 1 Configurations of the benchmarks: SLS on a $1,000 \times 2,500,000$ array, with the result located at index (350, 875000). UTS using the UTS T1 configuration, a geometric tree with a branching factor of 4 and a maximum height of 12; graph of size 164 million. SUD on a 25×25 board with 196 unsolved entries. NQK computes first 250 thousand solutions on board size of 15×15. UTP tree using the UTS T3 configuration, a binomial tree with a maximum height of 11; 700 goal nodes; and graph of size 13 million nodes. FLP on a 64×64 grid with 14 cells. TSP on 24 cities. KMC with 3 million points partitioned into 15 clusters. J2D using an array of size $5,000 \times 5,000$ with a block size of $1,000$. DLS on a $1,000 \times 2,500,000$ array, with the results located at index (100, 250000) and (350, 875000). CS on a $20 \times 20 \times 60 \times 15,000$ array, with the result located at index (8, 8, 24, 6000).

Benchmark Name	Acronym	Source (Eureka Pattern)	Description
Single Linear Search	SLS	Authors (Section 5.1)	Search for a single item in a 2D array.
Unbalanced Tree Search	UTS	UTS [30], (Section 5.1)	Search for a goal node in UTS trees which represent characteristics of various parallel unbalanced search applications.
Sudoku	SUD	Authors (Section 5.1)	Solving a Sudoku puzzle by exploring a game tree.
NQueens first K solutions	NQK	Authors (Section 5.2)	Find first K solutions to placing N queens on a chessboard such that no queen can attack any other.
UT Shortest Path	UTP	Authors (Section 5.4)	Trees from the UTS benchmark, adds edge weights to find shortest path to any goal node.
BOTS Floorplan	FLP	BOTS [12], (Section 5.4)	Compute the optimal floorplan distribution of a number of cells using branch and bound technique.
Traveling Salesman Problem	TSP	R. Wiener [46], (Section 5.4)	Solved using a branch and bound algorithm.
Jacobi 2D	J2D	Authors (Section 5.6 style)	Stencil computation with iterative convergence.
K-Means Clustering	KMC	Authors (Section 5.6 style)	An iterative refinement technique which converges to a local optimum.
Double Linear Search	DLS	Authors (Section 6.1)	Search for two items in a 2D array.
Composite Search	CS	Authors (Section 6.2)	Search for a single item in a multi-dimensional array.

JVM configuration flags[2]. The arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

Speculative Execution Benchmarks: The benchmarks are described in Table 1. The benchmarks include some of our motivating examples, search benchmarks, game puzzles, greedy algorithms, branch and bound algorithms, and a stencil computation. We present empirical evaluation of our implementation of the EuPM (`EU`) relative to variants that: (a) provide no support for early termination of `async-finish` tasks (`AF`); (b) use function-scoped cancellation points for termination of speculative `async-finish` tasks[3] (`FS`); (c) use exceptions for termination of speculative `async-finish` tasks[4] (`EX`); and (d) use `if` checks and `return` statements via cancellation tokens speculative `async-finish` tasks (`CT`). The results for execution time and productivity metrics are described below.

Execution Times Comparison: We compare the performance of the different `Eureka` patterns in the benchmarks. The comparison with the `AF` versions shows that most of these benchmarks benefit from speculation. In fact, in some of the benchmarks (e.g. SUD, TSP) the non-speculative variant does not complete execution. In other benchmarks, e.g. SLS, UTS, NQK, FLP, the non-speculative versions perform higher numbers of abstract operations (e.g. comparisons, arithmetic operations, nodes visited, etc.) which reflects in larger execution time compared to the speculative variants.

In general, the benchmarks SLS – J2D use a single eureka pattern and the `EX`, `CT`, and `EU` variants perform similarly. `EU` performs much better than the `FS` variant since the `EU` variant, like `EX` and `CT`, can trigger task cancellation even inside nested function calls. Overall, the

[2] `-XX:-UseGCOverheadLimit -Xmx65536m -XX:+UseParallelGC -XX:+UseParallelOldGC`.
[3] `if` checks and `return` happen only at the body of the `async`, not inside nested function calls.
[4] Our implementation minimizes overheads as it does not terminate worker threads, and it does not fill the stack trace of the abort exceptions.

Table 2 Benchmark execution time metrics, DNC means Did Not Complete inside 30 minutes. Except SUD, all the benchmarks had a coefficient of variation (CoV, ratio of the standard deviation to the arithmetic mean) less than 2 percent for the execution time of each variant. For SUD the CoV was about 10 percent for each variant.

Name	Execution Time (in seconds)					Ratio of Exec. Time				Abstract Operations ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF:EU	FS:EU	EX:EU	CT:EU	AF	FS	EX	CT	EU
SLS	58.37	17.70	16.61	16.71	16.85	3.46	1.05	0.99	0.99	2,476	845	798	800	806
UTS	15.89	8.81	5.94	5.81	5.76	2.76	1.53	1.03	1.01	1,571	512	444	446	437
SUD	DNC	5.52	5.53	5.48	5.72		0.96	0.97	0.96		146	148	142	152
NQK	24.90	3.33	3.86	3.20	3.96	6.28	0.84	0.97	0.81	1,711	216	210	212	216
UTP	2.95	2.73	2.58	2.37	2.48	1.19	1.10	1.04	0.96	233	233	189	189	189
FLP	38.35	30.25	7.83	7.94	8.04	4.83	3.79	0.98	0.98	688	523	232	233	231
TSP	DNC	1.51	1.18	1.19	1.11		1.35	1.06	1.07		857	839	839	754
KMC	15.22	12.26	12.32	12.56	12.44	1.22	0.99	0.99	1.01	1,125	916	916	916	917
J2D	16.35	13.01	13.21	13.04	13.10	1.25	0.99	1.01	1.00	1,125	902	902	903	903
DLS-AND	169.67	50.94	47.67	48.15	47.87	3.54	1.06	1.00	1.01	500	172	164	162	163
DLS-OR	169.00	4.65	0.53	0.53	0.54	315.17	8.66	0.99	0.99	490	15	2	2	2
CS	7.33	7.36	7.44	7.55	2.86	2.56	2.57	2.60	2.64	360	360	360	360	135
						>4.15	1.53	1.08	1.06	>1,027	474	434	433	409
						Geometric Mean				Arithmetic Mean				

Table 3 Productivity metrics for benchmark kernels. LoC was computed using `cloc` command while CC and DE were computed using the CodePro Analytix Eclipse plugin. LoC for common support code are not reported in the table, the arithmetic mean for support code LoC is 240.

Name	Lines of Code					Cyclomatic Complexity					Development Effort ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU
SLS	72	75	79	78	69	1.66	1.77	1.88	1.88	1.55	11.22	12.16	14.84	12.97	10.8
UTS	76	84	88	87	81	1.50	1.70	1.80	1.80	1.60	7.63	11.66	12.74	12.15	9.2
SUD	86	94	98	97	92	1.60	1.80	1.90	1.90	1.70	15.62	21.55	23.49	22.61	19.0
NQK	81	87	94	93	85	1.60	1.70	1.80	1.80	1.60	16.12	18.90	22.91	20.72	17.5
UTP	85	101	105	104	91	1.70	1.81	1.90	1.90	1.54	11.63	19.84	24.30	21.34	13.7
FLP	115	115	116	115	108	1.91	2.00	2.08	2.08	2.00	62.89	64.42	65.47	65.97	70.9
TSP	89	101	105	104	99	1.60	1.80	1.90	1.90	1.54	22.28	33.79	35.83	35.28	27.8
KMC	115	127	128	127	135	1.38	1.69	1.69	1.69	1.46	46.57	51.24	55.98	51.24	56.8
J2D	146	152	156	155	148	1.64	1.78	1.85	1.85	1.53	111.70	116.83	127.58	119.18	104.4
DLS	80	85	89	88	79	1.88	2.22	2.33	2.33	1.66	16.80	19.46	22.42	19.99	17.4
CS	108	131	139	138	107	1.61	1.70	1.82	1.82	1.61	39.37	73.43	87.35	84.67	43.7
A. Mean	96	105	109	108	99	1.64	1.82	1.90	1.90	1.62	32.89	40.30	44.81	42.37	35.6
%age of EU	-3.75	5.30	9.41	8.41		1.63	12.25	17.76	17.76		-7.49	13.33	26.02	19.17	

`EU` variants compete favorably with the other speculative variants (`EX` and `CT`). On most benchmarks, the `EU`, `EX` and `CT` variants perform within 5% of each other, both in terms of execution time and the number of abstract operations. This shows that our `EU` abstractions and different `Eureka` patterns do not add significant overhead in their implementations. Note that our implementation uses delimited continuations without modifying the VM; the performance of our implementation would be greatly improved by using native support for DeConts in the VM. Work by Stadler et al. [42] to provide such native support in a Java VM reported over two orders magnitude speedup on micro-benchmarks compared to a bytecode transformation approach. Additionally, we decided to exclude benchmarks that further highlight the limitations of the other approaches (e.g. inaccessible functions, user exception handlers) to allow a fairer comparison between all the approaches. These benchmarks would show our `EU` approach in "an even more" positive light.

The DLS benchmark uses binary composition of `EU Eureka`s and performs similarly to `EX` and `CT` confirming that no significant overhead is introduced by the composition. The CS benchmark is interesting as the hierarchical nature of the computation allows the `EU` variant to terminate other tasks searching on different leading indices. The `CT`, `EX`, and `FS` variants cannot implement such hierarchical information easily and end up doing redundant computation even after the goal element has been found. This causes them to perform as much work (in terms of abstract operations) as the non-speculative version and causing larger execution times than the `EU` version.

Productivity Metrics Comparison: The most commonly used software productivity metric is program size or lines of code (LoC) to compare programs that use the same language and coding standards. There are other quantitative evaluation techniques for productivity apart from measuring LoC. McCabe introduced the Cyclomatic Complexity (CC) metric [28] based on the control flow structure of programs. CC represents the complexity of the algorithm, and poorly designed solutions have high CC values. Halstead's metrics [19] are also well-known measure of software complexity. The Development Effort (DE) metric represents the effort required to convert an algorithm to an actual code in a specific programming language.

We report values for LoC, CC, and DE for all our benchmark kernels in Table 3. We compare the metrics for the `AF` variants with four variants (`FS`, `EX`, `CT` and `EU`) which implement different task cancellation strategies. Overall, the `EU` versions require less LoC, CC, and DE being at least 5%, 12%, and 13% better, respectively, than any of the other speculative variants. More importantly, the percentage improvements are even larger when compared to the closest performing speculative variants – `EX` and `CT`. In addition, as explained in Section 3, the `EU` versions do not suffer from any of the drawbacks compared to the other speculative methods. On average, the `EU` solutions for the kernels are only slightly larger than the `AF` variants requiring three extra LoC, some extra DE (7.5%), while the CC is actually smaller. This shows that, for the benchmark kernels, minimal effort was required to transform the `AF` versions into speculative versions using our proposed model. In particular, the comparatively low value of DE for the `EU` variant also reflects positively upon the usability of the `Eureka` API. In summary, the `EU` solutions are more productive to implement than the `FS`, `EX`, and `CT` variants in terms of all three productivity metrics.

9 Related Work

Kolesnichenko et al. provide a comprehensive classification and evaluation of task termination techniques [24]. C# natively supports interruptive cancellation by throwing exceptions, and since the release of TPL also cooperative techniques [25]. Python supports interruptive cancellation of non-started tasks via executors and terminative cancellation of already started ones [37]. Java supports interruptive cancellations natively [34]. Pthreads library supports both termination and interruption of threads [3].

Burton [4] and Osborne [32] have both worked with speculative computation before. Burton proposes a deterministic feature that has simple semantics, i.e. produces the same result as a sequential computation. Osborne uses numerical priorities to order computations [32], in his work task priorities propagate among dependent (sponsored) tasks. The eureka scope of tasks is determined when they are stated ahead of time in `OR` clauses or as branches of a conditional. Computation termination is via the cancellation token approach where a programmer manually checks termination in each function. Compared to our model, Burton and Osborne style speculative execution support only the parallel search eureka pattern.

Prabhu et al. [36] propose two language constructs to declaratively express value speculation opportunities. Their approach relies on speculating the value of a computation and executing possible future computations that consume this value in parallel with the producer of the value. Our approach does not rely on value speculation and does not need to deal with the rollback of side-effects from *mispredicted* consumer tasks. Instead, we use speculative tasks in the EuPM to support a multitude of EuSCs.

Leaving the system in an inconsistent state is one of the problems with preemptive termination approaches. MVM [9] and J-SEAL2 [2] solve this problem by introducing isolation containers to segregate the data operated upon by tasks. Tasks cannot directly

share objects, and the only way for tasks to communicate is to use standard, copying communication mechanisms. Containers communicate via synchronous `receive` operations to pass notifications. Termination is effected on isolation containers by other containers; termination kills all worker threads assigned to individual containers. Our approach minimizes overheads as it avoids copying data, killing threads, and communicating via synchronous operations. In addition, creating containers is an expensive operation whereas, in our approach, creating multiple eureka sub-computations is cheap as it is akin to creating a task.

Cilk allows speculative work to be terminated through the use of Cilk's `abort` statement [15] inside function-scoped inlets. Cilk does not provide guarantees of when child tasks will be terminated, in fact, child tasks can be spawned even after the execution of an `abort` statement. However, the main difference is that in the EuPM only a subset of tasks can be terminated in contrast to terminating all child tasks via Cilk inlets. Perez and Malecha show several methods for implementing `abort` as a library in the Cilk++ system [35] by mechanically translating programs into continuation-passing style. Like our approach, spawned computations periodically poll to determine if they should terminate. While this transformation is simple, the problem with it is that it is not modular because it changes the signatures of functions that use the `abort` mechanism. This breaks the possibility for separate compilation without explicit annotations specifying which functions should be compiled to work with inlets and `abort`.

Ada offers a statement, `abort`, which allows a task to make abnormal any other visible task [16]. The `abort` statement will stop execution of the named task by the time it reaches a synchronization point, e.g. `delay` statement that suspends the execution of a task for some units of time. Unfortunately, the use of `delay` statements (even those with delays of 0.0) can be expensive operations, as each delay statement forces the runtime system to perform a context switch. In our approach, a task cannot directly cancel another task, it influences cancellation by triggering eureka events. Also, our `check` construct does not force a task to context switch, making it much cheaper to implement.

Both MPI and OpenMP support task grouping and cancellation. MPI provides termination support via the `MPI_Abort` function that terminates an MPI execution environment [45]. This function call makes a best attempt effort to terminate all tasks in the group of the communicator. OpenMP 4.0 API [31], released in July 2013, supports features to terminate parallel OpenMP execution cleanly. Tasks can be grouped to support deep task synchronization, and task groups can be terminated to reflect completion of cooperative tasking activities such as search. Threads check at user-defined cancellation points if cancellation has been requested. The cancellation points must be lexically nested in the type of construct specified in the clause; i.e. we cannot `cancel` from inside a method call. Our approach poses no such limitation on where a task can request cancellation and where the user-defined cancellation points can be placed in the program.

Tahan et al. [43] also propose a cancellation policy for OpenMP similar to Ada's abort where a task can cause the cancellation of a group of tasks (possibly not belonging to the same group as the currently executing task). In our approach a task can request cancellation of other tasks belonging to the same group. Like our approach, however, their approach also causes child tasks to inherit the cancellation properties from the parent task. Unlike our approach, certain tasks can be *protected* from being cancelled even though they belong to a cancelled group, and the task cancellation scheme is based on interrupts and exceptions. We chose to avoid such protected tasks to avoid any confusion and to keep the EuPM clean and simple.

10 Summary and Future Work

We introduced the Eureka parallel programming model (EuPM) that simplifies the expression of speculative parallel tasks in search and optimization algorithms. We have demonstrated the power of the EuPM as a mechanism for codifying various parallel eureka patterns. The constructs we propose simplify writing EuSCs and improve programmer productivity. Our implementation shows that the EuPM can also be implemented efficiently, especially with the need to terminate tasks cooperatively. Our performance results on benchmarks show that our implementation performs close to manual hand-coded versions while being shorter and less complex to write. We believe that our implementation techniques of the EuPM can easily be ported to other languages as well. We are planning to extend support for further eureka patterns and providing dynamic task priorities in EuSCs.

Availability

A supplementary artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The artifact contains the source code of our library implementations of the different `Eureka`s and the different benchmarks used to simplify repeatability of all the experimental data. Public distributions of the `Eureka` implementation in Habanero-Java library are available for download at `http://wiki.rice.edu/confluence/display/PARPROG/HJ+Library`. The EuPM has also been taught in the introductory parallel programming class for second-year undergraduate students at Rice University (COMP 322). Additional documentation and code examples, are available in the course lecture and lab materials at `http://wiki.rice.edu/confluence/display/PARPROG/COMP322`.

Acknowledgments. We are grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper. We would also like to thank Suguman Bansal, Brad Chamberlain, Prasanth Chatarasi, Tom Hildebrandt, Siam Hussain, Deepak Majeti, Sri Raj Paul, Alina Sbîrlea, Dragos Sbîrlea, and Hamim Zafar for their feedback on early drafts of this paper.

References

1 S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of molecular biology*, 215(3):403–410, October 1990.
2 Walter Binder. Design and Implementation of the J-SEAL2 Mobile Agent Kernel. In *SAINT'01*, pages 35–, 2001.
3 Bradford Nichols and Dick Buttlar and Jacqueline Proulx Farrell. *Pthreads Programming: Chapter 4 – Managing Pthreads*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.
4 F. Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Trans. Computers*, 34(12):1190–1193, 1985.
5 Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Plenum Publishing Co., 2005.
6 Philippe Charles and et al. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005.
7 Liming Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *FTCS-25*, Jun 1995.
8 Jens Clausen. Branch and Bound Algorithms – Principles and Examples. *Parallel Computing in Optimization*, pages 239–267, 1997.

9 Grzegorz Czajkowski and Laurent Daynés. Multitasking Without Comprimise: A Virtual Machine Evolution. In *OOPSLA'01*, pages 125–138, 2001.

10 Olivier Danvy and Andrzej Filinski. Abstracting Control. In *LFP'90*, pages 151–160, 1990.

11 Iulian Drago and et al. Continuations in the Java Virtual Machine. In *ICOOOLPS'2007*, 2007.

12 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite. In *ICPP'09*, 2009.

13 Marc Feeley. Polling Efficiently on Stock Hardware. In *FPCA'93*, pages 179–187. ACM, 1993.

14 Mattias Felleisen. The Theory and Practice of First-Class Prompts. In *POPL'88*, 1988.

15 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI'98*, pages 212–223, 1998.

16 J. Goldenberg and G. Levine. Ada's Abort Statement: License to Kill. *Ada Letters*, IX(6):97–103, September 1989.

17 Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

18 Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS'09*, pages 1–12, 2009.

19 Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977.

20 Christopher T. Haynes and Daniel P. Friedman. Engines Build Process Abstractions. In *LFP'84*, 1984.

21 Ilya Grigorik. Untangling Evented Code with Ruby Fibers. `https://www.igvita.com/2010/03/22/untangling-evented-code-with-ruby-fibers/`, 2010.

22 Shams Imam and Vivek Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP'14*, 2014.

23 Shams Imam and Vivek Sarkar. Habanero-Java Library: a Java 8 Framework for Multicore Programming. In *PPPJ'14*. ACM, 2014.

24 Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. How to Cancel a Task. In *Proceedings of MUSEPAT'13*, pages 61–72. Springer, 2013.

25 Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *OOPSLA'09*, pages 227–242, 2009.

26 Wei-Ming Lin, Wei Xie, and Bo Yang. Performance Analysis for Parallel Solutions to Generic Search Problems. In *SAC'97*, pages 422–430, 1997.

27 Andrey Marochko. Exception Handling and Cancellation in TBB – Part II, May 2008.

28 T. J. McCabe. A Complexity Measure. *IEEE Trans. on Soft. Engineering*, 2(4), July 1976.

29 Oliver Kowalke. Introduction (Boost Coroutines). `http://www.boost.org/doc/libs/1_53_0/libs/coroutine/doc/html/coroutine/intro.html`, 2009.

30 Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In *LCPC'06*, pages 235–250, 2007.

31 OpenMP API, Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

32 Randy B. Osborne. Speculative computation in multilisp. In *LFP'90*, pages 198–208, 1990.

33 W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.

34 Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

35 Ruben Perez and Gregory Malecha. Speculative Parallelism in Cilk++, 2012.

36 Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe Programmable Speculative Parallelism. In *PLDI'10*, pages 50–61, 2010.

37 Python Software Foundation. concurrent.futures — Launching parallel tasks, August 2014.

38 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *RV'10*, pages 368–383, 2010.

39 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *ICFP'09*, 2009.

40 Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In *ICS'08*, pages 277–288, 2008.

41 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP'08*, 2008.

42 Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In *PPPJ'09*, 2009.

43 Oussama Tahan, Mats Brorsson, and Mohamed Shawky. Introducing Task Cancellation to OpenMP. In *8th Int'l Workshop on OpenMP*, IWOMP 2012, pages 73–87, June 2012.

44 The Jikes RVM Project. Threading and Yieldpoints. `http://jikesrvm.org/Threading+and+Yieldpoints`, 2007.

45 The Open MPI Project. MPI_Abort. `https://www.open-mpi.org/doc/v1.8/man3/MPI_Abort.3.php`, 2014.

46 Richard Wiener. Branch and Bound Implementations for the Traveling Salesperson Problem. *Journal of Object Technology*, 2(2), 2003.

47 Wolfram. Solve Optimization Problems with Speculative Parallelism, November 2008.