JCilk’s Support for Speculative Computation

I-Ting Angelina Lee
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA

1. Introduction

JCilk is a Java-based multithreaded language that extends the semantics of Java [Gosling et al., 2000] by introducing “Cilk-like” [Supercomputing, 2001, Frigo et al., 1998] parallel programming primitives. JCilk supplies Java with procedure-call semantics for concurrent subcomputations and, more importantly, integrates exception handling with multithreading by defining semantics consistent with Java’s existing semantics of exception handling.

JCilk’s strategy of integrating multithreading with Java’s exception semantics yields some surprising semantic synergies. In particular, JCilk supports an implicit abort mechanism as part of JCilk’s exception semantics: when multiple subcomputations are executed in parallel, an exception thrown by one subcomputation signals its sibling computations to abort. This implicit abort yields a clean semantics in which only a single exception from the enclosing try block is handled.

In ordinary Java, an exception causes a nonlocal transfer of control to the catch clause of the nearest dynamically enclosing try statement whose catch clause handles the exception. The Java Language Specification [Gosling et al., 2000, pp. 219–220] states,

“During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception.”

In JCilk, we have striven to preserve these semantics while extending them to cope gracefully with the parallelism provided by the Cilk primitives. Specifically, JCilk extends the notion of “abruptly completes” to encompass the implicit aborting of any side computations that have been spawned off and on which the “abrupt completion” semantics of the Java exception-handling mechanism depends.

This implicit abort mechanism allows speculative computation, which is required by some parallel search algorithms, such as branch-and-bound and heuristic search [Dailey & Leiserson, 2002]. In these algorithms, some computations may be spawned off speculatively, but are later found to be unnecessary. In such cases, one wishes to terminate these extraneous computations as soon as possible so that they do not consume system resources.

It turns out that the exception-based abort mechanism provided by JCilk is cleaner to code with than the inlet-based one provided by Cilk. Specifically, JCilk extends Java’s exception semantics to allow exceptions to be passed from a spawned method to its parent in a natural way so that the need for Cilk’s aborting constructs, inlet and abort, is obviated.

This paper presents how JCilk supports speculative computation with its exception-based abort mechanism. The paper first summarizes the basic syntax and semantics in JCilk, and then gives a motivational example of how the Queens puzzle with speculative computation can be implemented in JCilk.

The work described in this paper represents joint research with John S. Danaher and Charles E. Leiserson.

2. The JCilk language

The philosophy behind our JCilk extension to Java follows that of the Cilk extension to C: the multithreaded language should be a true semantic parallel extension of the base language. JCilk extends Java\(^1\) by adding new keywords that allow the program to execute in parallel. If the JCilk keywords for parallel control are elided from a JCilk program, however, a syntactically correct Java program results, which we call the serial elision of the JCilk program. JCilk is a faithful extension of Java, because the serial elision of a JCilk program is a correct (but not necessarily sole) interpretation of the program’s parallel semantics.

To be specific, JCilk introduces three new keywords —

\(^1\)Actually, JCilk extends the serial portion of the Java language, and it omits entirely Java’s multithreaded support as provided by the Thread class.
A parent method spawns a child method, which is accomplished by preceding the method call with the `cilk` keyword. Analogous to Cilk, in JCilk the keyword `cilk` has the same meaning in JCilk as they do in Cilk.

To illustrate how we have introduced these Cilk primitives into Java, first consider the JCilk program in Figure 1 and assume that no exception happens during execution. The method `f1` spawns off methods `A` and `B` to run in parallel in lines 4 and 5. After spawning `A` and `B`, the execution of `f1` continues to spawn off `C` in parallel in line 9 and call method `D` normally in line 10. Then the execution waits at the `sync` in line 11 until all the subcomputations `A`, `B`, and `C` have completed. When they all complete, assuming that no exception occurs, `f1` computes the sum of their returned values as its returned value in line 12.

What happens, however, when an exception occurs during execution? When an exception is thrown, JCilk “semisynchronously” aborts the side computations of the exception, which include any method that is dynamically enclosed by the `catch` clause of the `cilk try` statement that handles the exception. Consider the same program in Figure 1 again and assume that an exceptions occurs during execution. If `A` throws an exception after `f1` has spawned `B` and all its subcomputations are automatically aborted. Similarly, if `B` throws an exception while `A` is still executing, `A` and all its subcomputations are automatically aborted. In either case, once all computations dynamically enclosed by the catching `cilk try` have terminated (either by throwing exception of by aborting), the corresponding `catch` clause or `finally` clause (if any) is then executed. On the other hand, if `C` throws an exception, which is not caught anywhere within method `f1`, all `A`, `B`, and their subcomputations are aborted, and the exception is propagated up to `f1`’s parent, as the reason why `f1` is terminated.

This implicit abort is performed semisynchronously, that is, when a computation is aborted, the abort happens only after each `spawn`, `sync`, or `cilk try` statement. JCilk’s semantics for semisynchronous aborts simplify the reasoning about program behavior when an abort occurs, limiting the reasoning to those points where parallel control must be understood anyway. In addition, JCilk provides for aborts themselves to be caught by defining a new subclass of `Throwable`, called `CilkAbort`, thereby allowing programmers to clean up an aborted subcomputation.

### 3. The Queens problem

Figure 2 illustrates how the so-called Queens puzzle can be programmed in JCilk. The program would be an ordinary Java program if the three keywords `cilk`, `spawn`, and `sync` were elided, but the JCilk semantics make this code a highly parallel program. This JCilk program serves as an example of how speculative applications can be coded in JCilk in a relatively simple fashion compared to in Cilk or in Java.

This implementation uses a speculative parallel search and exploits JCilk’s implicit abort semantics to avoid extraneous computation. The program spawns many branches in the hopes of finding a safe configuration of the `n` queens. When such a configuration is discovered, some outstand-
public class Queens {
    private int n;

    private cilk void q(int[] cfg, int r) throws Result {
        if(r == n) {
            throw new Result(cfg);
        }
        for(int c=0; c<n; c++) {
            int[] ncfg = new int[n];
            System.arraycopy(cfg, 0, ncfg, 0, n);
            ncfg[r] = c;
            if(safe(r, c, ncfg)) {
                spawn q(ncfg, r+1);
            }
        }
    }

    public static cilk void main(String argv[]) {
        int n = Integer.parseInt(argv[0]);
        int[] cfg = new int[n];
        int[] ans = null;
        cilk try {
            spawn (new Queens(n)).q(cfg, 0);
        } catch(Result e) {
            ans = (int[]) e.getValue();
        }
        sync;
    }
}

Figure 2: The Queens problem coded in JCilk. The program searches in parallel for a single solution to the problem of placing n queens on an n-by-n chessboard so that none attacks another. The search quits when any of its parallel branches finds a safe placement. The method safe determines whether it is possible to place a new queen on the board in a particular square. The Result exception (which inherits from class Exception) is used to notify the main method when a result is found.

4. Project progress

We have developed a solid set of semantics that extends Java's exception mechanism to handle gracefully the parallelism introduced by spawn and sync. We have also built a working version of the JCilk system, including both the compiler and the runtime system, which supports the semantics that we have designed. The system implementation serves as an important foundation for JCilk's semantic design. Only after implementing the actual system, can we be certain that our semantic design is "reasonable" and not just some semantics on paper.

Several applications are developed in JCilk to demonstrate the expressive of JCilk’s features, including Queen’s puzzle, parallel alpha-beta search, and LU Decomposition. The next goal of the project is to improve the system performance and to integrate JCilk’s threading model with other modern language features in Java.

References


