DOI 10.12694/scpe.v18i3.1305
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 3, pp. 253–260. http://www.scpe.org © 2017 SCPE

A NOTE ON CORRECTLY GATHERING RESULTS FROM JR'S CONCURRENT INVOCATION STATEMENT

RONALD A. OLSSON*

Abstract. JR's concurrent invocation statement (*CoStmt*, for short) provides a form of collective communication. The thread executing a *CoStmt* can gather results from each of many invocations. This paper presents a problem that arises in using Java `List` structures in gathering such results. It presents incorrect and correct attempts at solutions and discusses the tradeoffs, including performance, among the correct solutions.

Key words: Concurrency, concurrent programming, JR programming language, concurrent invocation, synchronisation

AMS subject classifications. 68N15, 68N19, 68N20

1. Introduction.

1.1. Background. The JR concurrent programming language extends Java with a richer concurrency model [1, 2], which is based on that of SR [3, 4]. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process (thread) creation, rendezvous, asynchronous message passing, semaphores, concurrent invocation, and shared variables. These language features are provided by adding several new types and statements to the Java language. JR provides fairly transparent distributed programming (built using RMI, but hiding its details) and automatic program termination when the program has been detected to have quiesced. The JR implementation [2, 5] extends the Java implementation.

This paper focuses on an interesting problem that arises with the use of JR's concurrent invocation statement (*CoStmt*, for short) [5, 6]. The *CoStmt* can be viewed as providing collective communication between one thread and a group of others, for example, in applications such as initiating subparts of numerical computations, reading/writing a distributed database, and distributed voting schemes. Several concurrent programming languages and systems provide mechanisms to facilitate such communication, such as: MPI's [7] collective communication; and .NET's [8] delegates and asynchronous invocations. The general pattern is that one thread broadcasts data to a group of threads and then gathers back results from each in the group.

1.2. The problem and its challenges. This problem arose in practice when we were converting part of a JR program with a *CoStmt* from using an array to use an `ArrayList`. The specific problem arises when the thread gathering results of concurrent invocation needs to save the results in such a way as to preserve their order, which is fairly typical use of a *CoStmt*. Gathering into an array solves this problem easily, using the natural order provided by the array's subscripts. Gathering into a `List`, however, turns out to be more challenging. Code that gathers into a `List` that appears to be similar to code that gathers into an array does not always work. On the surface such code appears as though it should work, but these appearances are misleading.

Part of the problem involves issues of synchronisation in updating a `List`. Additional synchronisation can, in some cases, make a solution work, but it turns out that one correct solution requires no such additional synchronisation.

1.3. Outline. Section 2 introduces the *CoStmt* via examples that gather results into an array. It then presents attempts—incorrect and correct—for equivalent *CoStmts* using various kinds of Java `List` structures and discusses tradeoffs among the correct solutions, including the amount of required synchronisation. Section 3 describes the performance of the correct solutions and further explores the costs of synchronisation in the underlying Java `List` structures. Finally, Sect. 4 concludes the paper.

* Department of Computer Science; University of California, Davis; One Shields Avenue; Davis, CA 95616-8562 USA; (olsson@cs.ucdavis.edu).

2. Gathering results from *CoStmts*.

2.1. JR extensions and terminology. This paper uses only a small subset of the JR extensions to Java. In particular,

- JR provides threads[1] built on top of Java threads. Within the JR implementation, creating a JR thread includes creating a Java thread and setting up various structures used in implementing JR features such as quiescence detection and allowing invocations across physical machines (using RMI) [1, 2, 9].
- JR's *operation* abstraction generalises a method. An operation can be invoked in two ways and serviced in two ways, which provides flexibility in solving concurrent programming problems. However, this paper requires only the "method" style of operation invocation and servicing. Unlike a Java method invocation, a new JR thread is created to execute the method code while the invoking thread waits. (However, some optimisations allow the invoking thread to execute the method code directly [1, 2], but those optimisations do not apply for concurrent invocations from a *CoStmt*.)
- JR provides a concurrent invocation statement, which is introduced via examples in Sect. 2.2. However, this paper uses only a subset of the *CoStmt*.

TABLE 2.1
Declarations and other code used in the examples. (The declarations are used where needed; Copy-A and Init-L are used only when explicitly stated.)

Decl-A:	Declaration of array `A`.	`int [] A = new int[N];`

Decl-f:	Declaration of operation `f` and its code.	`private static op int f(int i) {`
		`  return i;`
		`}`

Decl-L:	Declaration of `ArrayList L`.	`ArrayList<Integer> L = new ArrayList<Integer>(N);`

Copy-A:	Copy array `A` to `ArrayList L`.	`for (int a: A) {`
		`  L.add(a);`
		`}`

Init-L:	Initialise `ArrayList L`.	`for (int i = 0; i < N; i++) {`
		`  L.add(null);`
		`}`

2.2. Example *CoStmts* using an array and their properties. A typical use of a *CoStmt* is as follows. Suppose we have an array as declared in Decl-A (Tab. 2.1) and want each of its `N` elements `A[i]` to be assigned the value of `f(i)`, where operation `f` is as shown in Decl-f (Tab. 2.1).

Co-1/AD in Tab. 2.2 shows a *CoStmt* that solves this problem. This *CoStmt* contains only one "arm" (as do all *CoStmts* in this paper). The arm contains a *quantifier* that indicates that the assignment to `A[i]` and invocation `f(i)` is to be done for the specified range of values of `i`. The thread executing a *CoStmt* (the "parent" thread) initiates all the specified invocations in parallel. A separate thread is created to evaluate each invocation of `f(i)`.[2] The parent thread then waits for the invocations to complete. When an invocation completes, the parent thread assigns the return value from the invocation to `A[i]`, i.e., the parent thread executes the assignments to `A[i]` one at a time. After all invocations have completed, the parent thread continues executing after the *CoStmt*.

[1] This paper uses the term "JR thread", although the other JR literature uses the traditional term "process" to represent a separate thread of control.

[2] In practice, each invocation of `f` should require sufficient computation to justify creating a new thread to evaluate it. Here, for simplicity of exposition, it does not.

TABLE 2.2

CoStmts *and wrappers used as examples. Our notation for referring to an example* CoStmt *includes a number indicating the order in which it appears in this table (or 0 if it is only discussed in the paper) and two letters: the first is 'A' for array, 'L' for* `ArrayList`*, 'S' for* `synchronizedList`*, or 'V' for* `Vector`*; the second letter indicates where the element is assigned: 'D' for directly in the assignment containing the invocation, 'P' for in the PPC, or 'W' for in the wrapper. An example wrapper is given the same number as its associated* CoStmt*.*

Co-1/AD:	`co (( int i = 0; i < N; i++ )) A[i] = f(i);`

Co-2/AP:	`int fi;`
	`co (( int i = 0; i < N; i++ )) fi = f(i) {`
	`  A[i] = fi;`
	`}`

Co-3/AW:	`co (( int i = 0; i < N; i++ )) invokeAndAssign( A, i );`

Wr-3:	`private static op void invokeAndAssign(int [] A, int i) {`
	`  A[i] = f(i);`
	`}`

Co-4/AD:	// same as Co-1/AD, but then copies `A` to `L` (Copy-A (Tab. 2.1))

Co-5/LP: (*incorrect*)	`int fi;`
	`co (( int i = 0; i < N; i++ )) fi = f(i){`
	`  L.add(fi);`
	`}`

Co-6/LW: (*incorrect*)	`co (( int i = 0; i < N; i++ )) wrapperAdd(L, f(i));`

Wr-6: (*incorrect*)	`private static op void wrapperAdd(ArrayList<Integer> L, int fi) {`
	`  L.add(fi);`
	`}`

Co-7/LW: (*incorrect*)	`co (( int i = 0; i < N; i++ )) invokeAndAdd(L, i);`

Wr-7: (*incorrect*)	`private static op void invokeAndAdd(ArrayList<Integer> L, int i) {`
	`  L.add(f(i));`
	`}`

Co-8/LP	// first initialise `L` (Init-L (Tab. 2.1)); then:
	`int fi;`
	`co (( int i = 0; i < N; i++ )) fi = f(i){`
	`  L.set(i, fi);`
	`}`

Co-9/LW:	// first initialise `L` (Init-L (Tab. 2.1)); then: `co (( int i = 0; i < N; i++ )) invokeAndSet(L, i);`

Wr-9:	`private static op void invokeAndSet(ArrayList<Integer> L, int i) {`
	`  L.set(i, f(i));`
	`}`

An arm of a *CoStmt* can also specify *post-processing code (PPC)*. The parent thread executes the PPC as each invocation completes; the arm's quantifier variables are accessible within the PPC. Thus, for example, Co-1/AD can be written equivalently using a PPC as in Co-2/AP, again insuring that the assignments to `A[i]`

are executed one at a time.

Each of Co-1/AD and Co-2/AP have the properties:

PropParallel: The invocations `f(i)` are evaluated (at least conceptually) in parallel.

PropSynch: The assignments to `A[i]` are synchronised, i.e., only the parent thread accesses `A` so no data races occur.

PropOrder: The assignments to `A[i]` give the desired order i.e., on completion, `A[i]` contains `f(i)`. Note, though, that assignments to `A` are *not* guaranteed to occur in the order of `i` (0, 1, 2, ...) due to the nondeterministic ordering of execution of the threads that evaluate `f(i)`.

Co-3/AW shows another *CoStmt*, which uses the wrapper Wr-3. It is like Co-1/AD and Co-2/AP in that it has the PropParallel, PropOrder, and PropSynch properties. However, by placing the assignment to `A[i]` within `invokeAndAssign`, it permits parallel assignments to different elements of `A` (whereas recall that Co-1/AD and Co-2/AP serialised these assignments).

2.3. Using an `ArrayList` instead of an array. We now consider a problem similar to that in Sect. 2.2. Suppose we have an `ArrayList` [10] as declared in Decl-L (Tab. 2.1). We want to write a *CoStmt* to assign it values as we did for the array in Sect. 2.2, i.e., element 0 contains `f(0)`, etc.

Of course, we could use Co-1/AD or Co-2/AP, and then copy from the array into the `ArrayList` (Copy-A (Tab. 2.1)), i.e., Co-4/AD.

However, some might find using an array only so it can later be assigned to an `ArrayList` to be aesthetically unpleasant and might be concerned about potential extra costs in doing so (discussed in Sect. 3). So, the problem becomes:

> Can we directly use an `ArrayList` within the *CoStmt*?

2.3.1. Incorrect `ArrayList` solutions. Our first attempt, Co-5/LP, is similar to Co-2/AP: it uses the PPC for adding `fi` to `L`.

Co-5/LP, however, is *not* equivalent to Co-2/AP as it does not provide PropOrder: the invocations of `f(i)` can complete in any order, so their values can be added to `L` in any order, not necessarily in order of `i`. We saw that behaviour in our experiments (Sect. 3): in some executions, some elements of `L` were not in the correct places.

Alternatively, we can use a *CoStmt* that assigns directly to the `ArrayList`, similar in style to Co-1/AD, such as Co-6/LW. Co-6/LW uses a wrapper operation, `wrapperAdd` (Wr-6), since what the *CoStmt* can invoke must be a JR operation and `ArrayList`'s `add()` is just a regular Java method.

However, Co-6/LW, is *not* equivalent to Co-1/AD or Co-2/AP as it does not provide PropParallel. Since `f(i)` appears as a parameter of the invocations of `wrapperAdd()`, they are evaluated serially, by the parent thread, before the invocations occur. (Co-6/LW also has other problems, similar to those presented for other *CoStmts* in the following discussion.)

The simple fix to Co-6/LW is to change the code to "wrap" not only the `add()` but also `f(i)`, e.g., Co-7/LW and Wr-7. This code now provides PropParallel. However, it does not provide PropOrder for the same reason as given for Co-5/LP (and with the same behaviour observed in experiments). Moreover, worse than Co-5/LP, `add()` in `wrapperAdd()` is not properly synchronised so invocations of `add()` that occur in parallel are not guaranteed to work, as per Fig. 2.1. In our experiments, the effect was that some elements were "lost": the overall size of the `ArrayList` was less than `N`, i.e., it was missing some elements.[3]

> "Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; *merely setting the value of an element is not a structural modification* [emphasis added].)"

FIG. 2.1. *Excerpt from the Java API for* `ArrayList` *[10].*

[3]That behaviour is what we observed with the Java implementation we used; such behaviour is *not* guaranteed by the Java API for `ArrayList` [10], e.g., program execution might lead to other results or a runtime exception.

To have Co-7/LW provide PropSynch, we can change from using just an `ArrayList` to use a Java Collection that does provide synchronisation. `synchronizedList` [11], e.g.,

```
List<Integer> L = Collections.synchronizedList(new ArrayList<Integer>(N));
```

or a `Vector` [12], e.g.,

```
Vector<Integer> L = new Vector<Integer>(N);
```

As now expected, our experiments show that, in either case, `L` has all of the desired `N` elements.

However, neither change provides PropOrder.

2.3.2. Correct `ArrayList` solutions. To have the code provide PropOrder, we take note of the emphasised part of Fig. 2.1. Co-8/LP now initialises the `ArrayList` so that all its elements are `null` (Init-L (Tab. 2.1)). That allows the PPC to use `set()` to place the element at the correct location in `L`.

This solution requires that `L` be just an `ArrayList`, i.e., it does *not* require `L` to be a `synchronizedList` or a `Vector` (although it would also work with either of those). The code is now not using any `ArrayList` method that is making a "structural modification"; it is just setting element values.

Symmetric to Co-3/AW, we can also use a wrapper, `invokeAndSet`, that both invokes first `f` and then `set()`, as seen in Co-9/LW and Wr-9.

Note how the correct `ArrayList` solutions require an exact bound (or at least an upper bound) on `N`. Such a bound is required, for example, so that the code has an `i` with which to `set(i, f(i))`. Needing `i` here is similar to needing an identifying index along with partial answers in the reply messages in other applications, such as pipeline sort [2].

3. Performance. This section presents performance comparisons in terms of execution times. The reported times are for only the critical part of program execution: the execution of the *CoStmt* (and any initialisation), not program startup or input/output.

3.1. Platforms. The data presented were obtained on a 4-core 4.00GHz system with 16 gigabytes of RAM running Fedora Core 24 Linux (4.10.10-100.fc24.x86_64). We ran the tests when the system was lightly-loaded. The data are based on the averages over several executions; variances were not significant. We also ran the tests on two other multicore platforms: a 3.60GHz system with otherwise identical hardware and software as our just-mentioned test system; and a 4-core 2.80GHz system with 16 gigabytes of RAM running Ubuntu 16.04 Linux (4.4.0-75-generic). The specific results differed, but they showed the same general trends. The specific versions of software that we used are JR 2.00607 [5] built with Java 1.8.0_60.

3.2. Test programs. The programs we tested were based on the *CoStmts* in Tab. 2.2. Specifically, we report on only the performance of the *correct* solutions. However, we observed that when the incorrect solutions ran properly their execution times were also close to the executions times reported for the correct solutions.

Table 3.1 and Fig. 3.1 present the data for those *CoStmts*. The data show little differences in the performances. The dominant cost is the `N` invocations from the *CoStmt*. Each such invocation creates a new JR thread, which is expensive: it includes creating a Java thread, plus additional overhead for implementing the additional JR features, e.g., those noted in Sect. 2.1. The work done by each thread is minimal, e.g., evaluating `f(i)`, whose computation is trivial.

The differences in whether the code uses an array, `ArrayList`, `synchronizedList`, or `Vector`, or uses PPC, wrappers, or neither makes no significant difference in performance in these *CoStmts*. However, one might prefer a particular *CoStmt* in Tab. 2.2 for stylistic reasons.

Similarly, in a more realistic program, where the cost of evaluating `f(i)` would be more than the cost of starting a thread, that cost would dominate the cost of potentially extra synchronisation.

3.3. Additional Java `List` tests. To compare the extra synchronisation costs of using Java's `synchronizedList` or `Vector` versus the non-synchronised `ArrayList`, we performed separate tests on a multi-threaded Java program. To focus on the cost of synchronisation on the various `List` structures, versus the cost of thread creation, we limited the number of threads. This program creates a list with `E` elements and

TABLE 3.1
Execution times (in milliseconds) on the CoStmts *in Tab. 2.2 over a range of number of invocations (*`N`*).*

Number of invocations (`N`)	Co-1/AD	Co-2/AP	Co-3/AW Wr-3	Co-4/AD	Co-8/LP	Co-9/LW Wr-9	Co-0/SW Wr-0	Co-0/VW Wr-0
1	61	61	64	63	59	61	60	60
2	62	61	62	63	63	62	61	62
10	61	60	61	64	61	60	61	61
20	62	62	63	65	62	62	65	63
100	93	93	96	95	93	95	98	93
200	138	137	139	137	137	134	146	137
1000	906	908	906	909	905	903	858	911
2000	4534	4543	4570	4536	4551	4556	4543	4548

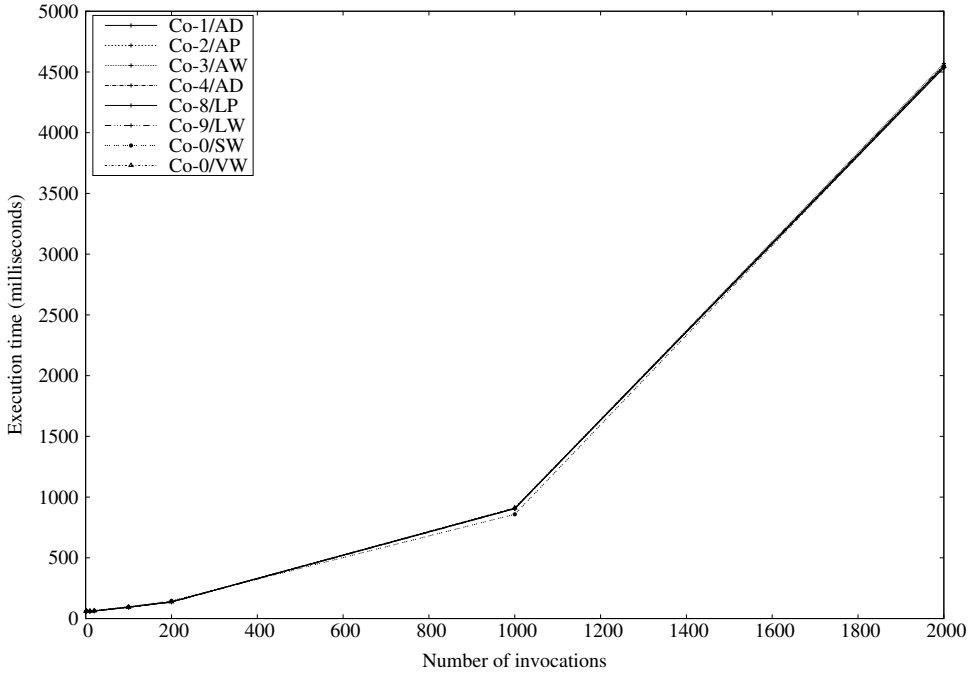

FIG. 3.1. *Execution times (in milliseconds) on the* CoStmts *in Tab. 2.2 over a range of number of invocations (*`N`*).*

then assigns values to each of the elements, much like the JR programs in Tab. 2.2. The program creates three threads that split the work: each thread is responsible for assigning values to (roughly) `E/3` elements in the list.

We tested two versions of this program for each kind of list: one version uses `add()` (similar to Co-7/LW) the other version uses `set()` (similar to Co-9/LW). The tests of the first version again showed that these solutions are incorrect for the same reasons given in Sect. 2.3.1. The tests of the second version confirmed it is correct and produced the data shown in Tab. 3.2 and Fig. 3.2. Note that these data are given in nanoseconds whereas the data in Tab. 3.1 and Fig. 3.1 are given in milliseconds.

The data show little differences in the performances of the three kinds of lists for the smaller values of `E`, i.e., less than 500 elements. However, after that point, the `ArrayList` version executes noticeably faster than the `synchronizedList` or `Vector` versions, presumably due to the extra cost of synchronisation. Also, after that point the `Vector` version executes a bit faster than the `synchronizedList` version.

The performance of these Java programs confirms the results discussed in Sect. 3.2 and seen in Fig. 3.1 and Tab. 3.1 for the JR programs. In particular, the relatively small differences in the costs of using an `ArrayList`, `synchronizedList`, or `Vector` are not noticeable within the JR programs.

TABLE 3.2
Execution times (in nanoseconds) of multithreaded Java `List` *programs over a range of number of elements (*`E`*).*

Number of elements (`E`)	`ArrayList`	`synchronizedList`	`Vector`
10	285409	300594	302091
50	304258	315736	305113
100	346149	355863	332241
500	543540	663671	548225
1000	819318	1185960	855099
5000	1632873	2566195	2025190
10000	2251727	4088558	3368276
50000	4786085	8395234	8381863
100000	6454248	13450121	12783344
500000	19583036	39482694	39359887

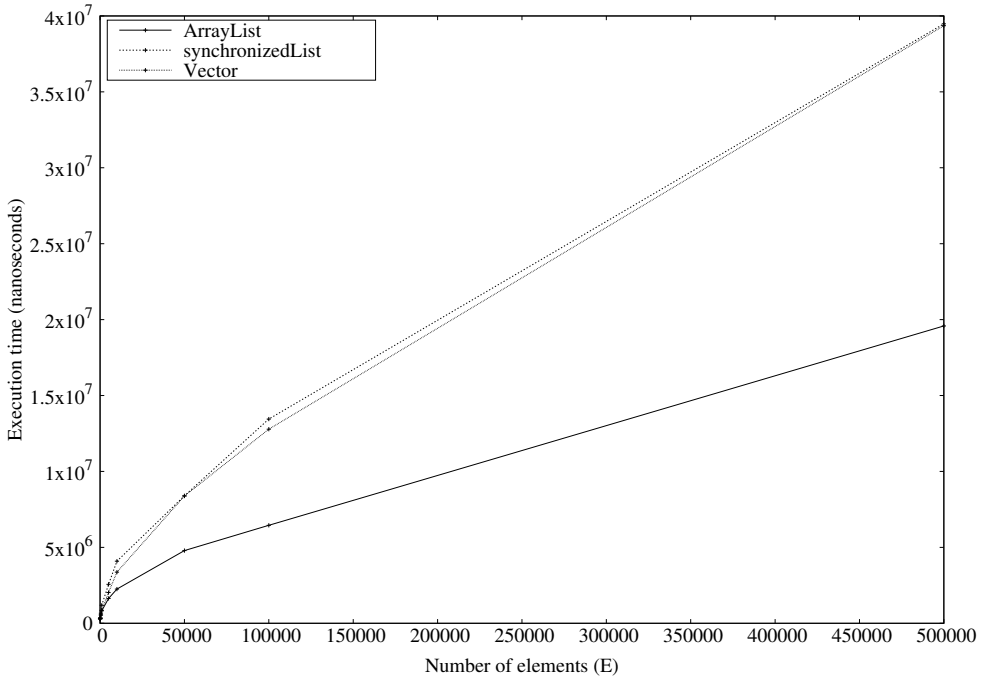

FIG. 3.2. *Execution times (in nanoseconds) of multithreaded Java* `List` *programs over a range of number of elements (*`E`*).*

4. Conclusion. This paper has presented a problem that arises in using Java `List` structures in gathering results from the *CoStmt*. It presented incorrect and correct attempts at solutions and discussed the tradeoffs among the correct solutions. Some of these solutions require less synchronisation. This paper also examined the performance of the correct solutions and further explored the costs of synchronisation in the underlying Java `List` structures.

The results show that, for this problem, the performances of the various correct *CoStmts* are roughly the same. Despite some requiring extra synchronisation, that extra cost is not significant compared to the larger costs inherent in the program.

The results comparing the costs of synchronisation in the underlying Java `List` structures and using initialisation and `set()` to provide the needed order can also be applied to regular Java threads programming and perhaps to programming in other concurrent languages and systems with similar features.

Acknowledgments. Nancy J. Wilson gave helpful feedback on the content and presentation of this paper. The anonymous reviewers carefully read this paper and gave helpful comments (and very promptly), which helped us improve this paper. The Editor-in-chief, Prof. Dana Petcu, responded promptly and encouragingly

to our queries.

REFERENCES

[1] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 578–608, May 2004.

[2] R. A. Olsson and A. W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer Academic Publishers, Inc., 2004. ISBN 1-4020-8085-9.

[3] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[4] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.

[5] JR language implementation, last accessed 2015-04-04 . `http://www.cs.ucdavis.edu/~olsson/research/jr/`.

[6] H. N. (Angela) Chan, E. Pauli, B. Y. Man, A. W. Keen, and R. A. Olsson. An exception handling mechanism for the concurrent invocation statement. In Jose C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, number 3648 in Lecture Notes in Computer Science, pages 699–709, Lisbon, Portugal, August 2005. Springer–Verlag.

[7] Message Passing Interface Forum, last accessed 2015-04-04. `http://www.mpi-forum.org/`.

[8] Programming with the .NET framework, last accessed 2015-04-04. `http://msdn.microsoft.com/en-us/library/aa720433(v=vs.71).aspx`.

[9] R. A. Olsson and T. Williamson. RJ: A Java package providing JR-like concurrent programming. *SOFTWARE–Practice and Experience*, 46(5):685–708, 2016.

[10] Class ArrayList<E>, last accessed 2017-04-15. `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

[11] Class Collections, last accessed 2017-04-15. `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`.

[12] Class Vector<E>, last accessed 2017-04-15. `https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html`.

Edited by: Dana Petcu
Received: May 2, 2017
Accepted: Jul 25, 2017