Implementation of PQL in Java 8 using ExtendJ

David Åkerman
D, Lund University, Sweden
dat12dak@student.lu.se

Yu Zhang
Lund University, Sweden
yu.zhang.7161@student.lu.se

Abstract

Effective use of parallelism can improve performance, but it is at the same time hard to get right. A solution to this problem is to use language specially designed to make parallelism easy, PQL is one such language. PQL is a query language embedded in Java which tries to make parallelism easier for the programmer by letting the underlying implementation take care how the computation is done. PQL has only been implemented in Java 6 and Java 9 has already been released. The PQL language can be implemented in the extensible Java compiler ExtendJ which is highly modular and more easily updated when a new version of Java is released. We have implemented a subset of PQL in ExtendJ and evaluated its performance. We found that our implementation was a bit slower than a parallel Java8 Streams in some benchmark, but has the same performance in other benchmarks.

1 Introduction

Today parallelism is a important concept to achieve high performance. Problem is that with traditional programming models programming solutions which utilize parallelism is not trivial, may introduce hard to find bugs like deadlocks and race conditions. Also it generally complicates the code which makes it harder to change later. A solution to this problem is to use another programming language which make it easy to utilize parallelism. Sequential computing is still important though and should still be possible in this language.
A answer is to embed a language which can handle parallel computations well in a language which can handle sequential computations. Examples of this is Parallel Language Integrated Query(PLINQ)[2] and Parallel Query Language(PQL)[5]. Both PLINQ and PQL are declarative query languages which is based on first order logic and let the programmer express computations in terms of what should be computed instead of how it should be computed. How the computation should be done is instead decided by the underlying implementation. This makes it possible for the implementation to do optimizations such as parallelization. PLINQ is a Microsoft solution which is embedded in .Net languages and PQL is embedded in the Java 6 language. Another solution that is not a query language is to do computations with streams were StreamJIT[6] and Java 8 streams are some examples. Instead of queries you modify a stream with a sequence of map, filter and reduce operations.
PQL was implemented directly in the javac compiler by modification of the already existing code base by Reichenbach[5]. Modification of the javac code is a problem because for each new release of the java language the new javac compiler needs to be rewritten. It is proposed that instead of rewriting the javac compiler for each new release, the ExtendJ compiler is extended with PQL instead. ExtendJ is an extensible Java compiler which lets you extend the Java language by changing the grammar, adding tokens and implementing semantic analysis with the Jastadd reference attribute grammar. In this project we implement a subset of PQL for Java 8 with ExtendJ. This implementation is evaluated by comparing the performance of this solution with three different types of code, ordinary for loops, Java8 Streams and the original PQL.

Course paper, EDAN70, Lund University, Sweden
January 15, 2018.

2 Language

PQL is a query language which is based on first order logic and use operators such as "and" and "or", as well as quantifiers such as "exists" and "forall". PQL reserve the keywords query, reduce, forall, exists and over. With PQL you can use query to create new collections by doing intersections and unions of already existing collections with additional conditions which filter out specific elements. Collections which PQL support is map, set and array. PQL also use the higher-order function reduce with the keyword reduce. At the current implementation the PQL language act as an extension to the Java language. It lets a Java-expression(JAVA-EXPR) be an PQL query (QUERY).The query has the production:

⟨*QUERY*⟩ ::= ⟨*QUANT-EXPR*⟩
  | id
  | ⟨*JAVA-EXPR*⟩
  | ⟨*QEXPR*⟩

A quant-expression has the production:
⟨*QUANT-EXPR*⟩ ::= ⟨*QUANT*⟩ ⟨*ID*⟩ ‘:’ ⟨*QUERY*⟩
  | query ‘(’ ⟨*MATCH*⟩ ‘)’ ‘:’ ⟨*QUERY*⟩
  | reduce ‘(’ id ‘)’ ⟨*ID*⟩ [over ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩

⟨*QUANT*⟩ ::= forall
  | exists

The quant expression forall will check if a variable fulfills a condition for all of its possible values and exists check if at least one possible value of the variable fulfill the condition. Both of these quant expressions will evaluate to a boolean value. A simple example of a forall quant-expression is:

```
forall x: 2*x == x + x
```

This expression will check if the variable x multiplied with 2 is is equal to x added with x for all possible values. This is of course always and will evaluate to the value true. A similar example for exists is:

```
exists x: x % 2 == 2
```

This will test if there exists a integer x divided by 2 with a remainder of 2. This is not true for any integer so it will evaluate to false.

A query has the non-terminal symbol MATCH and it has a production for each of the supported collections:

⟨*MATCH*⟩ ::= 'Set' '.' 'contains' '(' ⟨*ID*⟩ ')'
  | 'Map' '.' 'get' '(' ⟨*ID*⟩ ')' '==' ⟨*ID*⟩ [default ⟨*QUERY*⟩]
  | 'Array' '[' ⟨*ID*⟩ ']' '==' ⟨*ID*⟩

A concrete example of a query quant-expression is:

```
query(Set.contains(person)):
          (students.contains(person) ||
          teachers.contains(person))
          && person.country == India
```

This query will do a union of the set of students and the set of teachers and filter out every person which does not come from India. So in the end we will get a set of every student and teacher from India.

A reduce require a reduce function with a method signature "public static T r(T, T)". Other requirements of function is that it is associative, commutative and that T is unambiguous. To get more specific details about the reduce function check the original PQL report [5].

A reduce can be written like:

```
reduce(sum) tax: taxes.contains(tax) &&
tax > 1000
```

This reduce will evaluate to the sum of all taxes that is over 1000.

In PQL there is a specific expression called Q-expression which can use logical variable in the quant-expression and is without side-effects. Ordinary Java expressions can also be inside a quant-expression but they can not use logical variables. A Q-expression has the production:

⟨*QEXPR*⟩ ::= '(' ⟨*QUERY*⟩ ')'
  | ⟨*QUERY*⟩ ⟨*BINOP*⟩ ⟨*QUERY*⟩
  | ⟨*QUERY*⟩ instanceof ⟨*JAVA-TY*⟩
  | ⟨*UNOP*⟩ ⟨*QUERY*⟩
  | ⟨*QUERY*⟩ '?' ⟨*QUERY*⟩ ':' ⟨*QUERY*⟩
  | ⟨*QUERY*⟩ '.' 'get' '(' ⟨*QUERY*⟩ ')'
  | ⟨*QUERY*⟩ '[' ⟨*QUERY*⟩ ']'
  | ⟨*QUERY*⟩ '.' 'contains' '(' ⟨*QUERY*⟩ ')'
  | ⟨*QUERY*⟩ '.' id
  | ⟨*QUERY*⟩ '.' length
  | ⟨*QUERY*⟩ '.' size '(' ')'

The Q-expression is very similar to a ordinary Java-expression. Differences is that a Q-expression does not have method calls but instead have cases for some specific collection methods such as contains, size etc. <BINOP> is ordinary binary operations found in Java with the addition of implication as "=>" and UNOP is unary operations found in Java.

A feature in PQL is that the logical variable in a quant-expression can be either implicit or explicit typed.

⟨*ID*⟩ ::= id
  | ⟨*JAVA-TY*⟩ id

In case of a implicit type the range of values used is inferred by statically analyzing the <QUERY> in the quant-expression. When a explicit type is used the behavior is different depending on the type. If Int is used the conditions is evaluated for each of the possible 2^{32} values, the same goes for booolean and enum. For float, double and reference the condition is evaluated for each live value/object with the specific type on the Java heap. To get a more detailed description and some more examples check the original work[5].

3 Implementation

The implementation of PQL is largely based on the ExtendJ framework. Like other compilers, this implementation can be divided into five parts: lexical analysis, syntactic analysis, semantic analysis, code generation and optimization.
For the simplicity, we changed the grammar slightly and only implement a part of the PQL. For one thing, in the original PQL definition, the java expression is parallel with the quant expression and both the two kinds of expression belong to query, which means they should have the same priority. However, in our implementation, quant expression is implemented as a subset of java expression to avoid too much modification in ExtendJ. For another, our PQL can only support two sorts of quant expressions: exists expression and query expression with set.

3.1 Lexical Analysis

The main problem we need to solve is to eliminate the ambiguity caused by PQL keywords.
Consider the following PQL statement:

```
query (Set.contains(x)): x == 0
```

And the normal Java statement as below:

```
Set<String> s = Collections.emptySet();
```

The token "Set" may have different meaning when PQL is implemented based on Java, either a keyword or an implication of the Set interface. The solution to this ambiguity is to define a new lexical state in the scanner that has different actions than the default state. In detail, the scanner changes its state from the default state to query-state (the new state defined) when it meets the token "query" and alters from query-state to the default state after the ":" token. Accordingly, the token "Set" generates a keyword in the query-state while an identifier in the default state.

3.2 Syntactic Analysis

Firstly, several new classes are defined according to the PQL grammar.
We defined an abstract class *Query* to be a subclass of *Expr* and a class *QuantExpr* to be a subclass of *Query*. Although this seems redundant at this moment, it is necessary for the future extension. There are two subclasses of *QuantExpr* that are defined as below:

$$QueryQuantExpr : QuantExpr ::= Match\ Expr;$$
$$QuantQuantExpr : QuantExpr ::= CM\ Expr;$$

The *QueryQuantExpr* and *QuantQuantExpr* corresponding to the *query expression* and *exists expression* respectively. To complete the grammar, we defined the class *Match* and *CM* as well.

$$abstract\ Match;$$
$$SetMatch : Match ::= CM;$$

Currently, since we only support the query for sets, there is only one subclass of *Match*: *SetMatch*. However, it is extensible to support *MapMatch* and *ArrayMatch* in the future.

$$CM ::= Modifiers\ TypeAccess\ Declarator;$$

The class *CM* is similar to a variable declaration so we took usage of the $Modifiers$, *TypeAccess* and *Declarator* that are classes already defined in ExtendJ to make PQL more compatible to ExtendJ and easier for semantic analysis.
Secondly, new production rules are added to parse the PQL expressions.
For the *query expression*, the rule is:

$$query\_expr \ \rightarrow \ QUERY\ (\ match\ )\ :\ expression$$

In the rule above, "*QUERY*", "(", ")" and " : " are terminal symbols while "*match*" and "*expression*" are non-terminals symbols. Accordingly, the class *QueryExpression* contains two children: one is a match, another is an expression. The match will decide which kind of data type the query will create and the expression is the rule that the query will base on. When this rule is met, the parser will return an instance of *QueryExpression*.
Similarly, the production rule for *exists expression* is:

$$exists\_expr \rightarrow\ EXISTS\ cm\ :\ expression$$

In this rule, "*EXISTS*" and " : " are the keyword symbols. Meanwhile, a *cm* and an expression are component that will determine how the *exists expression* works. The non-terminal symbol *expression* that appears in both rules for *query expression* and *exists expression* is a normal java expression. *match* and *cm* are elements in PQL. To complete the context free grammar, we included the rules for *match* and *cm* as well.

$$match \rightarrow SET\ .\ CONTAINS\ (\ cm\ )$$
$$cm \rightarrow type?\ declarator$$

Since *cm* is similar to a java variable declaration, we reused the *type* and *declarator* from the original ExtendJ. Finally, to implant the PQL in ExtendJ, we added a production rule to $postfix\_expression$. Not only can $postfix\_expression$ derive from $primary$, $name$, $postincreament\_expr$ or $postdecrease\_expr$, it can also derive from $query\_expression$ now since $query\_expression$ should have the same precedence as these elements after our modification of the grammar.

3.3 Semantic Analysis

3.3.1 Type Analysis

Our quant-exppressions make use of ordinary Java-expressions instead of Q-expressions. Which means that many analyzes are already implemented in ExtendJ including Type Analysis. So with only some small additions we got several compiler checks for our quant-expressions.

3.3.2 Name Analysis

The *query expression* uses different lexical scope from the normal java expression because the variable in the expression part of a *query expression* can either be defined before this *query expression* or in the match part of this *query expression*. Therefore, when the name checker looks for declaration for a variable in a *query expression*, it should take the match part of this *query expression* as well.

3.3.3 Set Analysis

Because *query expression* need to have at least one set to iterate over at the start when it should filter the set. We simplify this compared to the original PQL by require that there is a call to the method contains in the expression of a query. By analyzing this call we can retrieve a set which we can iterate over. If no set is found we get a compiler error. Therefore, we defined a new attribute *getContainSet*() for the node *QueryQuantExpr* and *QuantQuantExpr*. This attribute will return a list of all the sets that used as $theSet.contains()$ in the expression part. Moreover, a collection attribute will throw an error to the error list if the attribute *getContainSet*() is null.

3.4 Code Generation

To generate code for PQL, the strategy is to transform the *quant expression* to an equivalent *stream expression* and then we let the already existing ExtendJ code generate Bytecode for the *stream expression*.
For a query expression, we use a parallel stream to iterate over a set and filter the elements in this set based on the expressions of the $query\_expression$. In detail, the main structure of the $stream\_expression$ will be $someSet.stream().paralell().filter(x$

someExpr). The *someSet* is one of the sets in the expressions of original *query expression* and *someExpr* is the expressions without the *someSet.contains*(). For example, the query expression below:

```
query (Set.contains(x)): s.contains(x) && x==0
```

will be translated to:

```
s.stream().parallel().filter(x -> x==0)
          .collect(Collectors.toSet())
```

Both of the two expressions above are to create a new set composed of the elements that belong to the set *s* and equal to 0.

In the transformation, what we need to know is the set to be iterated, the remaining expressions and the type of elements in the set. Firstly, for the set to be iterated, the first set in the attribute *setAllContains*() will be picked out. Secondly, for the remaining expressions that will be used as the filter conditions, we create a new attribute called *exprRemoveOneContain*() that we use to replace the contain method call for the set we iterate over with the boolean literal *true* (because it will always be true anyway). Thirdly, because type inference in lambdas is not fully supported, the type of lambda expression need to be specified with a cast. Attribute *predicateType*() will use the elements in *setAllContains*() to get the callers of *contains*(). The caller is either a method call or a set and the element type is accessible by the attribute *type*() of the method call or the set. For a exists expression we have a similar approach as query but we use the stream method anyMatch instead of filter.

4 Evaluation

4.1 benchmarks

There are four benchmarks that we use to evaluate the performance of PQL: webgraph, threegrep, setnested and setexists. The first three of them are was also used in the performance evaluation of the original PQL implemented in Java6. The fourth benchmark is a new one. We ran the benchmark on a Intel core i7 2.9GHZ.

- webgraph

```
Set<Webdoc> documents;
documents = Generator.documents;
result =query(Set.contains(Webdoc doc)):
  documents.contains(doc)
  && exists link:
     doc.outlinks.contains(link)
  && exists link2 :
     link.destination
        .outlinks.contains(link2)
  && link2.destination == doc;
```

  Webgraph carries out a complex computation. It will find webdocuments which link to webdocuments which link to the first webdocument. So webdocs that links to itself through another webdoc.

- threegrep

```
result = query(Set.contains(byte[] ba)):
  data.contains(ba)
  && exists j:
     ba[j] == ((byte) '0')
     && ba[j + 1] == ((byte) '1')
     && ba[j + 2] == ((byte) '2')
     && index.contains(j);
```

  Threegrep is simpler than webgraph. It will search for the byte array that contains "123" in a series. We did a little change to it because the original version does not have *set.contains*() in exists part which is a must in our implementation. But the function of the query remained the same.

- setnested

```
result = query(Set.contains(int x)):
  generatedSet1.contains(x)
  && x < 10
  && generatedSet2.contains(x);
```

  Setnested calculate the intersection of two large data sets. It is easy to understand that this leads to much computations for large sets.

- setexists

```
result = exists int x:
  generatedSet1.contains(x)&&
  generatedSet2.contains(x)&&
  generatedSet3.contains(x)&&
  generatedSet4.contains(x)&&
  generatedSet5.contains(x)&&
  generatedSet6.contains(x)&&
  generatedSet7.contains(x)&&
  generatedSet8.contains(x)&&
  generatedSet9.contains(x)&&
  generatedSet10.contains(x)&&
  x < 10;
```

Setexists is to check whether there is an element that ten large data sets have in common and that is lower than 10.

4.2 Evaluator

Besides our PQL, we used five other evaluators to run the benchmarks and compared the results with each other.

- Manual for loop
  What a PQL expression can calculate can also be done by for loop equivalently. This evaluator use for loops with a single thread.

- Parallel for loop
  This use for loop as well, but use multithreading with 4 threads.
- Stream
  This use Java8 Streams to do the computations.
- Parallel Stream
  This use Java8 parallel Streams. Only the outermost stream is parallel. Use 4 threads. A bit different than the streams which we generate in our PQL implementation which all will be parallel streams.
- Original PQL in Java6
  This is the PQL implemented in Java6 that does not use the stream. It use 4 threads. Was not used for setexists.

4.3 Results

The plots below show the results of different evaluators with different benchmarks. Each column corresponds to one of the evaluators and the bar of our PQL implementation is in red while the others are in blue. The height of each bar means the average time to complete. According to Andy Georges[3] non-determinism due to JIT compilation, thread scheduling and garbage collection makes Java performance hard to quantify. So to get a reliable result we run the calculations 60 times, ignore the time for the first 10 iterations and take a average of the time for the 50 following. There is a vertical black line on the top of each bar which represents the confidence interval of the average result(in some cases to small to be visible).

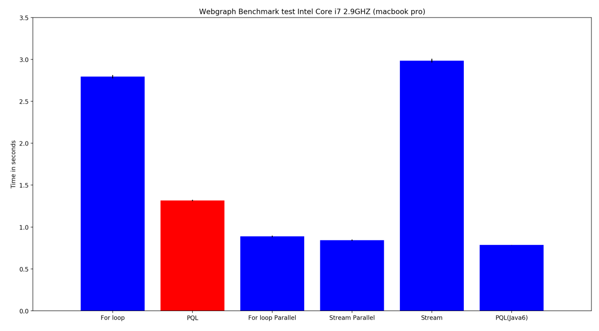

Figure 1. The performances of evaluators on webgraph

From this picture, our PQL outdoes the manual for loop that is the baseline. However, it is a little slower than parallel for loop, parallel stream and the PQL in Java6. The difference between the performance of our PQL and parallel stream is caused by the many parallel streams which cause some overhead.

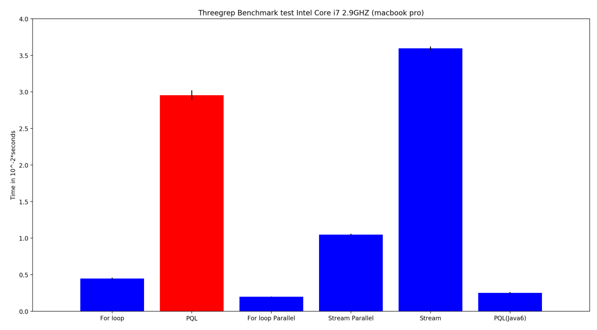

Figure 2. The performances of evaluators on threegrep

In this case, we can see that evaluators which use streams is the slower bunch. A possible explanation can be that the streams do not evaluate the conditions in the more optimal way. As before the overhead of using to many parallel streams makes our PQL slower.

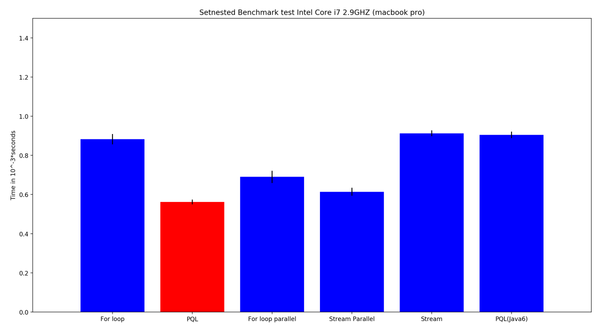

Figure 3. The performances of evaluators on setnested

There is many computations in this benchmark due to the large sets used but the logic for the benchmark is pretty simple. This benchmark only need one stream which means that parallel stream and PQL should generate the same code. As can be seen the confidence interval is a bit bigger. So our PQL has the same performance as parallel streams in this benchmark. Parallel stream seems to perform better overall. This benchmark is not as intensive as the other benchmarks before, which may expalin the sub par performance of the original PQL. The original PQL assumed that using several threads would not gain anything, so it only used 1 thread.

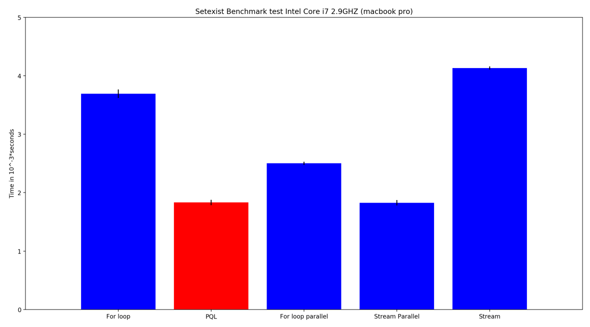

Figure 4. The performances of evaluators on setexists

As the benchmark before only one stream is used. This result is pretty similar to the previous benchmark as well.

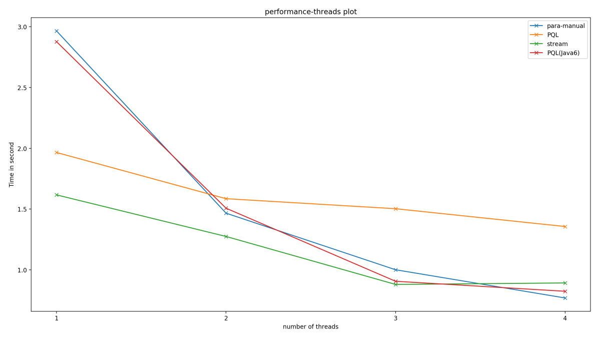

Figure 5. The performances of evaluators on webgraph with different number of threads

In this graph we see performance of evaluators for the webgraph benchmark with different number of threads. In the case of 1 thread the evaluators use the code of the parallel version but with only one thread.

5 Related work

The original work by Reichenbach [5] is most related to our work. In that work they designed the PQL language, implemented it in Java 6 and evaluated the performance. Our work has a smaller scope and only implement a part the language. Other difference is that we implemented the language as a extension to ExtendJ unlike the original work which was implemented by modifying the Javac compiler. Many of the benchmarks we use in the evaluation are the same as in the original work but we compare the implementation to other solutions for example in the original work they evaluate the performance on mysql and haddoop and in our work we evaluate the performance of Java Streams. As far as solutions that tries to solve the same problems as PQL the biggest contenders are Java Streams and PLINQ for C#. Unlike PQL Java Streams are a library in Java instead of a language. Java Streams lets you express computations with a chain of method calls on a stream object. The difference between how parallelization is handled with Java Streams is that the programmer need to use parallel stream as well as use a collection that can handle concurrent access to get parallel computations. PQL will automatically parallelize computations. Another advantage of PQL is that it will automatically reorder filter conditions at runtime to get better performance, this is not done with Java streams. PLINQ is also query language like PQL but PLINQ has syntax that closely resemble SQL with syntax like "select", "from", etc. PLINQ also require the programmer to specify which collections traversal is done in parallel[2]. Other solution that use the approach of using streams is StreamIt [6] and StreamPI[4]. StreamIt is programming language which was implemented as a extension to a Java compiler and use messages sent with a FIFO-queue through filters and streams to do computations. StreamPI is a library for Ada and C++ and use a similar approach as StreamIt.

6 Conclusion

We have implemented a subset of the PQL language as a extension to the ExtendJ compiler. Our version of PQL is able to do intersection of sets as well as filter sets with several conditions with the additional requirement that the expression inside a query contains a set calling the method contains. The quantifier exists was also implemented. Our implementation transform each query into the corresponding Java parallel Stream, which later is transformed to the correct byte code by the compiler. Our evaluation results show that our PQL implementation is a bit slower than Java Streams, the original PQL implementation and parallel version with for-loops. That our implementation is slower than Java Streams can be explained by the fact that we parallelize each of our streams which will create many threads which each add extra overhead that slow down the computations. Our work show that it is possible to implement subset of the PQL language in ExtendJ, but we think that most the PQL frontend could be implemented as well. Extending ExtendJ with the rest of PQL language could be done as a future work. Generating immediate language code for PQL(PQIL) that can be used by the runtime introduced in "A Backend Extension Mechanism for PQL/Java with Free Run-Time Optimisation"[1] to get better performance could also be done.

Acknowledgments

First we would like to thank Christoph Reichenbach which helped us throughout the project. Also we would like to thank Jesper Öqvist for showing us alternative solutions when we encountered bugs in ExtendJ.

References

[1] Hilmar Ackermann, Christoph Reichenbach, Christian Müller, and Yannis Smaragdakis. 2015. A backend extension mechanism for PQL/Java

with free run-time optimisation. In *International Conference on Compiler Construction*. Springer, 111–130.

[2] J Duffy and E Essey. 2007. Running queries on multi-core processors. *MSDN Magazine* (2007), 133–142.

[3] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.

[4] Jingun Hong, Kirak Hong, Bernd Burgstaller, and Johann Blieberger. 2012. StreamPI: a stream-parallel programming extension for object-oriented programming languages. *The Journal of Supercomputing* 61, 1 (2012), 118–140.

[5] Christoph Reichenbach, Yannis Smaragdakis, and Neil Immerman. 2012. *PQL: A Purely-Declarative Java Extension for Parallel Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 53–78. https://doi.org/10.1007/978-3-642-31057-7_4

[6] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 49–84.