PQL: A Purely-Declarative Java Extension for Parallel Programming

Christoph Reichenbach[1], Yannis Smaragdakis[1,2], and Neil Immerman[1]

[1] University of Massachusetts, Amherst
[2] University of Athens, Greece
{creichen,yannis,immerman}@cs.umass.edu

Abstract. The popularization of parallelism is arguably the most fundamental computing challenge for years to come. We present an approach where parallel programming takes place in a restricted (sub-Turing-complete), logic-based declarative language, embedded in Java. Our logic-based language, PQL, can express the parallel elements of a computing task, while regular Java code captures sequential elements. This approach offers a key property: the purely declarative nature of our language allows for aggressive optimization, in much the same way that relational queries are optimized by a database engine. At the same time, declarative queries can operate on plain Java data, extending patterns such as map-reduce to arbitrary levels of nesting and composition complexity.

We have implemented PQL as extension to a Java compiler and showcase its expressiveness as well as its scalability compared to competitive techniques for similar tasks (Java + relational queries, in-memory Hadoop, etc.).

1 Introduction

Parallelism is here to stay. Parallel hardware has already transitioned from niche architectures to mainstream computing. Power and latency trends (of electronics, as well as of other foreseeable physical processes) dictate that the computer industry shift permanently to parallel processing, instead of trying to improve on traditional single-core sequential designs. Programming parallel computers, however, is a formidable challenge. Various forms of parallel hardware have been around for decades, and generation after generation of programmers have been unable to utilize such hardware fully and easily through traditional programming models: Although there are well-known parallel algorithms, the base algorithmic thinking in computer science is sequential. Even worse, it is not the case that we can "start from scratch" and disregard sequential computation. A pure parallel future is unlikely. Fast sequential processing is the greatest advantage that digital computers hold over massively parallel natural computers, such as the human and animal brain. Thus, it seems inevitable that we are heading towards a future where we will need to program both sequential and parallel algorithms in a unified manner.

Our work advances parallel programmability through a unified sequential-parallel programming model, reified in a language design we call PQL/Java. We use a general-purpose programming language (Java) as the substrate and extend it with PQL: a declarative, logic-based sublanguage (based on first-order logic operators). Any program expressible in PQL will be automatically parallelized. In fact, the expressiveness of PQL is explicitly limited so as to allow efficient parallelization.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 53–78, 2012.

For a preliminary example of PQL, consider a simple program fragment:

```
int[] arr = query (Array[x] == y): range(1, 1000).contains(x) && y == x*x;
```

The above Java declaration uses a PQL expression to initialize an **int** array variable. The PQL expression starts by stating the form of the result: it will be an array (PQL keyword Array is one of three possible at this position, the others being Map and Set) mapping x to y. The body of the query specifies that y is the square of x, for x between 1 and 1000. ("range" is a library method that produces sets.) PQL will parallelize the evaluation, if deemed profitable, and split it among the available processors. (We give this and several other very short examples only language illustration: parallelization will not help for such simple expressions and small data amounts.)

The distinguishing feature of PQL is that it is transparently integrated in Java yet fully declarative, without any order dependencies between clauses. The New Oxford Dictionary of English defines "declarative" in the context of Computing as "denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed". In other words, a program or language is declarative when it specifies *what* needs to be computed but not *how*. The "how" can be highly variable and the language implementation has a lot of choice in this decision.

To see what declarativeness means in our context, consider what is perhaps the closest conceptual relative of PQL: the .Net PLINQ facility for parallel queries [9]. PLINQ is also a parallel query language with explicit syntactic support (inside .Net).[1] PLINQ and PQL differ in a myriad of language design choices—e.g., logic-based sentences (forall/exists clauses) vs. relational queries (select–from–where clauses). But the deepest difference is that PQL is fully declarative, thus allowing far more optimization and transformation of the parallel query code, but also limiting what can be expressed in a query. Consider a query that combines two data structures: a set, premiumcustomers, and a map, cust2orders, returning a new data structure that maps every customer in the set to their high-value orders (e.g., above a value of 1000). Both PQL and PLINQ can easily express such queries. In the case of PQL, we have:

```
Map m = query (Map.get(cust) == order):
        premiumcustomers.contains(cust) &&
        cust2orders.get(cust) == order && order.amount > 1000;
```

Similarly, in PLINQ one might write (adapting to C# idioms and structures):

```
var m = from cust in premiumcustomers.AsParallel()
        from order in cust2orders[cust].AsParallel()
        where order.amount > 1000
        select new { cust, order };
```

The difference is that the implementation of the PLINQ query has fewer degrees of freedom than the implementation of the PQL query. The programmer needs to specify which traversals are done in parallel. Also, the traversals are pre-ordained: the system will iterate over all elements in set premiumcustomers and then over all elements in each set of orders for the given customer. In contrast, the PQL query offers no guarantee or

[1] Strictly speaking, the syntax extensions to VB and C# are for LINQ, the sequential querying interface, and PLINQ requires no extra support.

even indication of how parallelism is applied or the order of traversal. The PQL implementation is free to reorder the query clauses in many ways, since all PQL expressions are guaranteed side-effect-free. (PQL queries can contain arbitrary Java expressions that refer to Java program variables, but not to PQL "variables", i.e., can be evaluated just once for the entire query.) The system may decide to iterate over all elements of the cust2orders map first, or over all elements of the premiumcustomers set, or even over all objects of type Customer that exist on the heap, even if they are not guaranteed to be members of premiumcustomers. The latter will not be cost-optimal for this query, but either of the former two traversals may be, depending on the sizes of the data structures. Similarly the implementation of the query may choose to partition (and parallelize the traversal of) either the premiumcustomers set, or the cust2orders map, or even the traversal of all heap objects.

PQL can express many parallel tasks, but *its main strength is for generalized, arbitrarily nested map-reduce-like relation manipulations*. Indeed, the language is explicitly designed to combine logical queries and reduction operators. The query language design targets a specific level of expressiveness, in order to enable parallelization. The inspiration comes from complexity theory. "Descriptive complexity" [11] is the sub-area of complexity theory that matches logical languages with complexity classes. In terms of descriptive complexity, first-order logic over finite structures is a language that can express exactly the problems that can be optimally parallelized, i.e., solved in constant time by a CRAM (Concurrent Random Access Machine)—a theoretical abstraction of a parallel machine—using a polynomial number of processors. Of course, this highly theoretical view ignores important practical overheads and constraints (e.g., when finding the minimum of n elements, we do not want to perform n^2 comparisons in practice, even if these are in parallel). Still, the theoretical expressiveness class serves as a good guideline for our design, and we can classify PQL precisely as a *first-order* query language.

Generally, the main contributions of our work are as follows:

- We present a new approach to parallel programming, consisting of an embedding of a fully declarative query language inside a general-purpose language.
- We discuss the embodiment of our approach in a specific language setting and detail its essential features for expressiveness and optimization. Although one can discern high-level similarities of our PQL/Java language with others (e.g., database query languages embedded inside general purpose languages), our need for tight integration of the two language models creates unique demands and opportunities at both the language design and implementation level.
- We present performance measurements of PQL/Java for sample tasks to showcase its areas of strength and implementation scalability. The results validate the ease with which simple declarative tasks can exploit parallelism, reaching the performance of manually optimized code.

2 Language Illustration

We begin with a description of the PQL/Java language, with several examples interspersed for illustration.[2]

[2] Language specification and implementation are available at `http://creichen.net/pql`

2.1 Language Constructs Overview

At the high level, PQL is primarily a first-order query language. This means that it can be viewed as a first-order logic, with the usual boolean connectives ("and", "or", etc.) and quantifiers ("forall", "exists"). As in every first-order language, the main functionality is defined as specialized predicates and functions that can be used in this logic. Additionally, PQL has an extra-logical component: it adds the ability to aggregate query results in more powerful ways than allowed by the logic ("reduce" them).

In more concrete terms, PQL defines the keywords **query**, **reduce**, **forall**, **exists**, and **over**. It further re-purposes existing Java constructs, including many operators (such as &&, ||) and some invocation-like expressions (such as set.contains(element)). In this idea it follows the Java Query Language JQL [16]. To integrate with Java, these constructs assume the meaning defined in this document in source files that contain the import statement:

```
import static edu.umass.pql.Query;
```

In the absence of this statement, the syntax and semantics of a PQL/Java program are identical to those of a regular Java program.

Syntactically, PQL/Java extends Java by allowing any Java expression (*JAVA-EXPR*) to be a query (*QUERY*). (We show the full syntax of PQL later, in Figure 1, but explain it here incrementally.) A query, in turn, follows the production:

$$\mathit{QUERY} ::= \langle \mathit{QUANT\text{-}EXPR} \rangle \mid \mathsf{id} \mid \langle \mathit{JAVA\text{-}EXPR} \rangle \mid \langle \mathit{QEXPR} \rangle$$

(We inherit from Java the usual non-terminals *JAVA-EXPR*, for Java expressions, *JAVA-TY*, for Java types, and id , for Java identifiers.) That is, a query may be a quantifier expression (*QUANT-EXPR*) that quantifies one or more *logical variables* (e.g., forall x, y : a[x] > b[y]), a single identifier that references such a logical variable (such as x or y in the above example), or one of two unquantified expressions: an arbitrary Java expression (which may contain side effects but cannot use logical variables—i.e., variables declared inside the PQL query) or a Q-Expression (*QEXPR*), which may use logical variables and sub-queries but no side effects. Since Java expressions may contain PQL queries, it is possible to nest multiple queries in the same expression, though these must not share variables.

Quantifier Expressions. Quantifier expressions take one of the following forms:

$$\begin{aligned} \mathit{QUANT\text{-}EXPR} ::= {} & \langle \mathit{QUANT} \rangle \ \langle \mathit{ID} \rangle \ \text{':'} \ \langle \mathit{QUERY} \rangle \\ & \mid \textbf{query} \ \text{'('} \ \langle \mathit{MATCH} \rangle \ \text{')'} \ \text{':'} \ \langle \mathit{QUERY} \rangle \\ & \mid \textbf{reduce} \ \text{'('} \ \mathsf{id} \ \text{')'} \ \langle \mathit{ID} \rangle \ [\textbf{over} \ \langle \mathit{ID\text{-}SEQ} \rangle \ ] : \ \langle \mathit{QUERY} \rangle \end{aligned}$$

The first form of quantification is universal or existential quantification: *QUANT* may be either forall or exists. Such an expression has boolean value, true or false. The second form of quantification, a *container query*, constructs maps, sets, or arrays. The third and final form is a general-purpose reduction operation.

Universal and Existential Quantifications. Universal or existential quantification extends over an identifier *ID*, which can explicitly declare a type:

$$\mathit{ID} ::= \mathsf{id} \mid \langle \mathit{JAVA\text{-}TY} \rangle \ \mathsf{id}$$

For now, consider an example of the second form:

```
forall int x : x == x
```

This tests whether all x that are of type **int** are equal to themselves. This particular test should always evaluate to **true**. Similarly,

```
exists int x : x*x == -1
```

will test whether there exists an integer x whose square is equal to -1; this test will evaluate to **false**. (Of course, the system has no way of knowing this fact statically, hence the query will be evaluated in parallel over all **int**s.)

We refer to the logical variable occurring in the *ID* construct as the *query variable*. If a Java type (*JAVA-TY*) is present, the query variable is *explicitly typed*, otherwise the range of values for the variable is inferred. These two cases behave differently. Informally, the difference is that for queries

```
/* A */ forall int x : rel[x] > 0
/* B */ forall x : rel[x] > 0
```

the compiler will infer the static type for case B, and also infer that it should only consider values for x that occur in the domain of rel, whereas for case A it will consider all 2^{32} possible **int** values for x, regardless of the size of rel. This topic is discussed in more detail in Section 2.3.

Container Queries. A container query has the syntax

query '(' ⟨*MATCH*⟩ ')' ':' ⟨*QUERY*⟩

where a *MATCH* is one of the following:

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

The first of the above productions then constructs a set, as in the following example:

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

This would construct a set of integers containing precisely the number zero. (The use of Set.contains in the syntax is an allusion to the method by the same name in the Java standard API Set interface, and similarly for Map.get.)

The second construction builds a map:

```
query (Map.get(x) == y): range(1, 10).contains(x) && y == x*x
```

This would construct a map of all numbers from 1 to 10 to their squares. range(1, 10) here is a logical constant and a Java expression, denoting a set of all integers from 1 through 10. By contrast, y == x*x is a PQL subexpression: both x and y are logical variables. Note that the above does not provide mappings for numbers outside the range. For example, index 0 of the generated map will be **null**.

Maps may contain a default clause. For instance:

```
query(Map.get(x) == y default -1): range(1, 10).contains(x) && y==x*x
```

This would construct the same map as above, except that all numbers outside of the range 1 through 10 are mapped to −1.

Our third construct builds arrays. For example,

```
query (Array[x] == y): range(1, 10).contains(x) && y == x*x
```

is the same as our map construction without defaults, with one exception: the missing array index (0) is filled in with the default value for the relevant type (i.e., 0 for integers). Thus, this will construct an 11-element array containing $0, 1, \ldots, 81, 100$. Any attempt to define the array at a negative offset raises an exception.

Reductions. The last kind of quantifier expression in PQL is a reduction, which follows the syntax below:

reduce '(' id ')' ⟨*ID*⟩ [**over** ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩

(*ID-SEQ* denotes a comma-separated sequence of *ID*.) The first parameter to a reduction is always a reduction operator, such as the built-in sumInt, sumLong and sumDouble operators. Consider:

```
reduce (sumInt) x : myset.contains(x)
```

This will sum up all numbers contained in myset after coercing them to **int**. The type of a reduce expression is the type of the value being reduced over (e.g., **int** for a sum of Java primitive integers).

Sometimes reduction requires additional free variables. We obtain these using the keyword over:

```
reduce (sumDouble) x over y : set.contains(y) && x == 1.0 / y
```

This will sum up the inverses of all numbers contained in the container set.

In later sections we will see all built-in reductors as well as how to provide user-defined ones.

QUERY ::= ⟨*QUANT-EXPR*⟩ | id | ⟨*JAVA-EXPR*⟩ | ⟨*QEXPR*⟩
QUANT-EXPR ::= ⟨*QUANT*⟩ ⟨*ID*⟩ ':' ⟨*QUERY*⟩
| **query** '(' ⟨*MATCH*⟩ ')' ':' ⟨*QUERY*⟩
| **reduce** '(' id ')' ⟨*ID*⟩ [**over** ⟨*ID-SEQ*⟩] : ⟨*QUERY*⟩
QUANT ::= **forall** | **exists**
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** '(' ')'
BINOP ::= '||' | '&&' | '|' | '&' | '^' | '%' | '*' | '+' | '−' | '/' | '>' | '<'
| '<=' | '>=' | '!=' | '==' | '<<' | '>>' | '>>>' | '=>'
UNOP ::= '!' | '~' | '−'
MATCH ::= 'Set' '.' 'contains' '(' ⟨*ID*⟩ ')'
| 'Map' '.' 'get' '(' ⟨*ID*⟩ ')' '==' ⟨*ID*⟩ [**default** ⟨*QUERY*⟩]
| 'Array' '[' ⟨*ID*⟩ ']' '==' ⟨*CM*⟩
ID ::= ⟨*ID*⟩ | ⟨*JAVA-TY*⟩ id
ID-SEQ ::= ⟨*ID*⟩ | ⟨*ID-SEQ*⟩ ',' ⟨*ID*⟩

Fig. 1. PQL/Java syntax

Q-Expressions. A Q-expression (non-terminal *QEXPR* in Figure 1) has essentially the same syntax as regular Java non-side-effecting expressions, with the exception of

method calls, which are not supported in general (though we do borrow method call syntax for a number of set and map operations). Q-expressions can freely refer to logical variables, and form the basis of parallel computations in PQL. All familiar Java operators have the same meaning inside a Q-Expression (including emulating the Java exception behavior). The only new operator is =>, denoting logical implication: 'a => b' is equivalent to '(!a) || b', with ! being logical negation.

There are some slight differences in how operators can be used with different types. Q-expressions of the form q.get(x) or q[x] are equivalent. Both denote a map or array lookup (depending on the static type of q) and evaluate to the element indexed by x. For example, myArray.get(3) for an integer array myArray would obtain the 4th element of the array. This operation raises the same exceptions as regular Java array accesses might raise. Q-expressions of the form q.f are projections that obtain the contents of field f. Field f must be accessible from the context in which the query originates according to the rules of reflective field access in Java (i.e., the field may be private and in a different class, but field access via the Java reflection API must be permitted). Finally, Q-expressions of the form q.length or q.size() are equivalent. Both evaluate to the size of an array, java.util.Collection, or java.util.Map, as determined by the static type of q.

2.2 Examples and Expressiveness

Before we discuss more advanced language features and semantics, we present examples of useful queries, to establish the usage model more firmly.

Consider the following prototypical map-reduce task [6]: identifying a three-character word in a set of strings. We here represent the strings as arrays of bytes (akin to the assumptions of Dean and Ghemawat [6]) and store them in an array of arrays, data_array. We compute the set of arrays that contain the string of interest ('0', '1', '2') with the following PQL query:

```
result = query(Set.contains(byte[] ba)):
           exists i: data_array[i] == ba
            && exists j: range(0, RECORD_SIZE − 3).contains(j)
              && ba[j] == ((byte)'0') && ba[j+1] == ((byte)'1') && ba[j+2] == ((byte)'2');
```

(Here, RECORD_SIZE is the number of bytes in all strings. We could equivalently use ba.length or ba.size(), which are evaluated at run-time, but we choose to keep the Dean and Ghemawat scenario of having static knowledge that we can exploit in the query.)

The PQL runtime automatically parallelizes this query, as it deems appropriate. For instance, the implementation may iterate sequentially over the contents of data_array in search of an appropriate component array ba, but then search each ba in parallel for a matching index, j. In later sections we use this query for illustration and describe how we translate it into our intermediate language and optimize it.

For a more complex example consider an adaptation of a common map-reduce motivating scenario [7]. An application keeps track of entities called Pages and Sites. A Page object uniquely maps to a Site (i.e., every page has a unique site, while a site "owns" pages) and Pages can refer to other Pages (i.e., every page is mapped to a set of other pages). This information is captured by regular Java maps and sets, namely two data structures "Map<Page,Site> page2site" and "Map<Page,Set<Page>> refersTo". Imagine that the programmer wants to implement the following functionality: for each page p,

count the number of sites that own pages that refer to p. This is captured by the following PQL query:

```
Map<Page,Integer> result =
  query(Map.get(p) == i):
    i == reduce (sumInt) one over Site s:
      one == 1 &&
      (exists Page pReferrer:
        page2site.get(pReferrer) == s && (refersTo.get(pReferrer)).contains(p));
```

In words, the program text says: compute a map from each page p to an integer i, so that i is the number of sites s (count one for each site) that own a page (and possibly more than one) that refers to p.

The query is automatically optimized, parallelized and executed by multiple processors. We can see some of the optimization reasoning in intuitive terms. The runtime system will first identify that there is a loop "exists pReferrer" over Page objects, another loop over Site objects (using variable s, over which we reduce), and implicitly a loop over variable p of type Page (which appears in the result). Thus, the query can certainly be evaluated by enumerating all n^3 combinations of values for p, s, and pReferrer. We can do better than that, however, since these values are related. The current pReferrer object is enough to index all relations used in the query (page2site and refersTo) and bind the values of p and s. To retrieve only relevant pReferrer objects, the system can partition the refersTo data and assign a portion to each processor. The results of each processor's computation will then be combined and finally reduced.

To see high-level optimization reasoning in more depth, consider another example over the same relations, page2site and refersTo. We would like to compute for every site the number of pages it owns that have outside references. This is accomplished by the following query:

```
Map<Page,Integer> result =
  query(Map.get(s) == i):
    i == reduce (sumInt) one over Page p:
      one == 1 && page2site.get(p) == s &&
      (exists Page pReferrer:
        page2site.get(pReferrer) != s && (refersTo.get(pReferrer)).contains(p));
```

For this example, an efficient evaluation will likely not start by traversing either instance of relation page2site. Examining an element of this relation does not lead to an efficient indexing of the other two relations involved in the query. (This is because the page2site map is efficient for retrieving sites given a page, but not vice versa.) Instead, a good evaluation order would start by enumerating relation refersTo and using the values of pReferrer and p that it obtains to index into the two instances of relation page2site. Furthermore, the expression "page2site.get(pReferrer) != s" does not really bind variable s: examining tuples of page2site for a given pReferrer tells us what s is *not*. Therefore, the optimal join order (for reasonable assumptions of relation sizes) is to start with refersTo, proceed to the first instance of page2site and then to the second. We see that an automatic optimizer is highly desirable even for small queries. For larger queries, it quickly becomes invaluable and relieves the programmer of the obligation to think about evaluation details, instead concentrating on the specification of the desired task.

In general, the optimizer needs to find the evaluation strategy (i.e., program transformations and join order) that binds all variables in the result tuple with the minimum lookup cost and space requirements for intermediate results. Any iteration over the elements of a type or a relation whose size exceeds a pre-defined threshold can be parallelized for efficiency: all that is required is a partitioning of the relation and assignment of each partition to the appropriate processor. A welcome property is that the problem of parallelization becomes *easier* the larger the size of the data involved in a query. Ideally, a single relation is partitioned and the rest of the evaluation logic (i.e., joining the rest of the logical conditions) is executed sequentially by the processor assigned to the partition.

Overall, PQL is quite expressive and allows arbitrary combinations of queries. Effectively, every query that one can express in relational algebra or SQL can also be expressed in PQL (since PQL includes full first-order logic plus aggregation operators, in the form of built-in reducers), although the difference in the structure of the query can be significant. In theoretical terms, this is exactly the class of queries that can be parallelized optimally, i.e., executed in constant time if the number of processors is large (but still polynomial relative to the input size) [11, Theorem 14.9, 5.27].[3] However, one should be careful in interpreting complexity theory results in a practical setting: although an algorithm expressed in such a query language can be parallelized optimally, this is useless if expressing the algorithm in the language greatly grows the cumulative work to be done by all processors together, e.g., from $\Theta(n)$ to $\Theta(n^2)$.

2.3 Beyond Basics

To complete our informal description of the language, we next discuss some important design issues: types, exceptions, our notion of equality, and user-specified reductors.

Types. PQL type checking generally imitates Java, with some exceptions. Types are implicit, unless annotated explicitly. As we have seen, logical variables can either be declared 'with type' (as in forall **int** i : ...) or 'without type' (as in forall i : ...). These declarations specify different semantics. The explicitly typed variant has the obvious semantics, for example, forall **int** i : i == i will loop over all 2^{32} integer values and check that they are equal to themselves. At runtime, explicitly typed variables conceptually iterate over all *viable* values of their type τ, where *viable* is defined as follows:

- If τ is an enumeration, the viable values of τ are all members of the enumeration, as per java.util.EnumSet.allOf.
- If τ is an ordinal type such as **int** or **boolean**, the viable values of τ are all possible values for τ permitted by the Java programming language.
- If τ is a floating-point type, i.e., **float** or **double**, then the viable values for τ are all the values of that type that exist in live objects on the Java heap.[4]

[3] Strictly speaking, this bound makes unrealistic assumptions with respect to the complexity of merging results (especially for reduction operations). Still, the practical approximation of such theoretical models typically incurs a $log(n)$ slowdown factor, where n is the input size, which is perfectly acceptable as a bound for parallelization purposes.

[4] Support for iterating over floating point and reference values requires a runtime with either heap traversal functionality (through a VM extension) or load-time code rewriting. We show that finding objects by type on the heap can be efficient in our earlier DeAL system [13], though our current PQL implementation does not yet replicate this feature.

- Otherwise, τ is a reference type, and the viable values for τ are all the objects of that type that exist in live objects on the Java heap.

By contrast, a logical variable without explicit type is subject to *domain inference*. This means that the variable's type and bounds are inferred from the body of the quantification. In terms of static types, we infer the type of a query variable as the least upper bound of all the types it can assume, eagerly defaulting to java.lang.Object at the least precise. In terms of runtime values, domain inference computes the domain of relevant relations. For instance, for an array a, the domain of an index variable (i.e., an x used in an expression a[x]) is the set 0 to a.length – 1 (inclusive). We use domain inference only on expressions where such a binding is intuitive and obvious—i.e., for predicates set.contains(x), array[x], and map.get(x). Domain inference extracts all syntactic occurrences of such subterms for quantified variables, including any dependencies: for example, when processing

```
forall x: forall int[] a: b.contains(a) && a[x] > 0
```

we must make sure to extract all viable a in order to determine bindings for b. When there are multiple matching subterms, such as

```
forall x: a[x] > 0 && b[x] > 0
```

the domain is the union of all possible domains, in this case the union of the index domains of a and b.

It is a static error whenever there is no such domain. In practice, this occurs precisely whenever the user omits an explicit type for a quantified variable x and in the body of the quantification this variable never appears as an index, key, or set element, or any such element, or only occurs in a context that depends on x itself, such as x.contains(x).

Exceptions. Exceptions in a query body are propagated to the outside. The language guarantees no order in which exceptions are delivered.
Queries such as

```
query(Map.get(int a) == int b) : a == 1 && range(1,2).contains(b)
```

(which compute multiple mappings for a map key) are invalid and raise a query failure through the edu.umass.pql.AmbiguousMapException.

PQL performs boxing/unboxing conversions implicitly, and failed conversions (i.e., attempts to unbox **null**) raise an exception.

Equality. We allow both the == operator and the = operator for equality comparison (albeit at different operator precedence, following the Java language definition).

Equality is reference equality, except in the following two situations:

- Equality between strings is always equality via equals (value equality).
- Objects in a map or set are considered equal under the terms of the dynamic map or set type. For example, keys of a java.util.HashMap are equal iff they are equal in terms of equals.

User-Defined Reductions. We allow user-defined reductions. A reduction operator r must have the following properties:

- r must be a static method with signature **public static** T r(T, T). The type T must be compatible with the values being reduced and determines the result type of the reduction.
- The type T must be unambiguous. This requirement is relevant when r is overloaded and type inference cannot determine a unique T.

- r must be associative. We distribute the reduction phase by having each core run reductions on part of the data, then merge the results of individual cores. Hence, our runtime system partitions reductions based on what hardware the program is run on. If r is not associative, we may get wrong results.
- r must be commutative. This is not technically required by our current implementation, as it merges the data in order. However, future implementations may choose other reduction orders.
- r must have a neutral element. If T is a reference type, the neutral element is always **null**. For primitive types, r must have a special-purpose Java annotation that specifies the neutral element.

User-defined reduction operators can violate both the parallelization properties (since their execution is sequential and can take arbitrarily long, although multiple are run in parallel) and the declarativeness of PQL. The language blindly trusts that such reductions respect the above stated properties, and chooses an evaluation plan based on this assumption.

3 Implementation and Optimization

Our current prototype implementation of PQL consists of a front-end compiler (integrated with the Oracle `javac`). Our implementation generates ASTs directly and leaves it to the `javac` backend to generate bytecode (which the JIT compiler may optimize further). The runtime support is currently entirely library-based, requiring no VM changes. However, specialized VM support can enable higher levels of optimization in the future. (E.g., by storing fields in random-access tables instead of contiguous objects, it may be more profitable to perform a query using such tables than using other participating relations.) In order to anticipate different back-ends in the future, and also to isolate the front-end language from back-end optimization techniques, we have introduced an intermediate language, PQIL. PQIL is a complex IL—here we discuss its essence and design rationale.

PQL is translated into PQIL and optimized using relational optimization techniques (similarly to an SQL query optimizer), with emphasis on interfacing with Java data and on parallelization. The minimal independent code unit in PQIL is a "join", which can be either a primitive join or a control structure. Control structures allow the sequencing of joins (i.e., conjunction) or combination of all their alternatives (disjunction). Consider the example query of Section 2.2 for identifying a three-character word in a set of strings. We repeat the query below for ease of reference:

```
result = query(Set.contains(byte[] ba)):
           exists i: data_array[i] == ba
             && exists j: range(0, RECORD_SIZE − 3).contains(j)
               && ba[j] == ((byte)'0') && ba[j+1] == ((byte)'1') && ba[j+2] == ((byte)'2');
```

This query is translated to the following PQIL representation:

```
Reduce[SET(?ba): ?result]: {
  JAVA_TYPE<byte[]>(?ba);
  ARRAY_LOOKUP_Object(?data_array; ?i, ?ba);
  ARRAY_LOOKUP_Byte(?ba; ?j, '0');
```

```
    ADD_Int(?j, 1; ?i8);
    ARRAY_LOOKUP_Byte(?ba; ?i8, '1');
    ADD_Int(?j, 2; ?i10);
    ARRAY_LOOKUP_Byte(?ba; ?i10, '2');
    INT_RANGE_CONTAINS(0, (RECORD_SIZE − 3); ?j);
}
```

The above is a reduction over the result of a series of joins that we consider in conjunction. Question marks as prefix indicate variables (named and temporary), all other identifiers are constants of one form or another. As can be seen, PQIL contains predefined predicates for all primitive expressions of the PQL language, turning evaluation of Q-expressions into relational joins. There are some 100 predefined predicates in total, covering types (e.g., JAVA_TYPE<**byte**[]> above), arithmetic (e.g., Add_Int, BITOR_Int), comparisons (e.g., LTE_Int), array/set/map lookup operations (CONTAINS, INT_RANGE_CONTAINS, ARRAY_LOOKUP_OBJECT), and more.

At the outermost level, the translation of a PQL query always contains a reduction operation. Operations forall and exists, as well as constructing maps, sets, etc. are all translated into special-purpose reduction operators. In the above PQIL code, the first line indicates the reduction: we compute a set of all ba, which we write into a result. To compute ba, we consider the body: we locate all ba that satisfy the conjunction of all conditions specified below:

- "JAVA_TYPE<**byte**[]>(?ba);": correct type.
- "ARRAY_LOOKUP_Object(?data_array; ?i, ?ba);": contained in data_array at index i (a variable that has no further purpose).
- "ARRAY_LOOKUP_Byte(?ba; ?j, '0');": at index j, ba has the character '0'.
- "ADD_Int(?j, 1; ?i8);": j+1 = i8.
- "ARRAY_LOOKUP_Byte(?ba; ?i8, '1');": at index i8, ba has character '1'.
- "ADD_Int(?j, 2; ?i10);": j+2 = 10.
- "ARRAY_LOOKUP_Byte(?ba; ?i10, '2');": at index i10, ba has character '2'.
- "INT_RANGE_CONTAINS(0, (RECORD_SIZE − 3); ?j);": j is contained in the range 0 through RECORD_SIZE − 3.

Here, RECORD_SIZE − 3 is a Java expression— RECORD_SIZE is not a query variable. Just as any other Java expression (however complex), it is a constant from the perspective of the query. Our query execution mechanism ensures that we compute the Java expression's value only once and cache it for the rest of query execution. (Recall that we make the assumption that queries do not have side effects, which is true modulo a few remaining effects "by design" such as OutOfMemoryExceptions, and user-defined reductors, which may violate our assumptions.)

The above list captures all constraints we want to have on ba, but the order of joining the 8 relations is crucial for performance. The order is determined by "access path selection", a standard optimization from the database literature [14], to estimate the cheapest way to join the predicates together. The overall process involves a number of steps. We begin with domain inference, which in this case has no effect: no more restrictions on the values of the existential variable can be inferred. Next, we determine dependencies between joins and eliminate unused variables (such as i in our example, transforming ARRAY_LOOKUP_Object(?data_array; ?i, ?ba); to

ARRAY_LOOKUP_Object(?data_array; _, ?ba);). Finally, we compute an access plan, i.e., the order in which the components of the reduction's join should be executed. There are several concerns in selecting this sequence. For instance:

- We reason about indexing: in which order do we bind our variables? Section 2.2 gives high-level examples of this challenge. In our IL, we write J(!x) to indicate that J binds variable x. After access path selection, there must be no free variables remaining.
- We avoid relations that cannot be traversed in the current implementation. For instance, in the above PQIL block, the predicate JAVA_TYPE<**byte**[]>(?ba); cannot form the beginning of the traversal: we have no way to enumerate all **byte**[] objects on the program heap without VM support, even if this were the most efficient way to evaluate the query. Therefore, the predicate can only appear at a position in the sequence where variable ba has been bound.
- Even if a relation can be traversed, it may not be parallelizable. For parallelization, each in-memory join object can expose an interface that allows (depending on the join itself) reductions such as our set computation to perform random access into the join. Queries that can be executed in parallel get a significant bonus during access path selection.
- We translate the query access plan into nested Java loops for later execution inside the VM (after JIT compilation and VM optimization of the bytecode). In practice, the outer loop is often parallel, hence it is important to have it be over a relation large enough to be profitably partitioned. It is also important to ensure good locality (e.g., accesses to consecutive, related data) for the computations in inner loops.

We represent each join in our intermediate language as a join object. There are three classes of join objects: primitive joins (such as ARRAY_LOOKUP_Object), composite joins (e.g., the above block of atomic joins, or the reduction), and "custom joins", which represent the custom generated code in the form of nested Java loops that we just described. This design gives us flexibility since it isolates the optimization logic from the runtime system implementation.

Note that our current implementation makes all scheduling decisions statically. In general, this may not be optimal; there are cases where dynamic information can make a significant difference between picking one option or another. Such dynamic information concerns mainly the size of relations and the distribution of values in a map or set (i.e., the likelihood of that a key will return a value, which determines the selectivity of a join). For example, when joining two maps over their key, it is preferable to iterate over the smaller map in the outer loop and perform lookups on the bigger map. PQIL has explicit primitives %SELECT_PATH and %SCHEDULE to designate that the sequencing of joins in a block is to occur at a later phase and what information it can use. We currently do not use the dynamic access path selection facility because it only works with interpreted execution of our intermediate language (i.e., not with the nested loops execution model).

The PQIL implementation also integrates several optimizations. These include simple optimizations, such as elimination of redundant joins (e.g., two occurrences of the same primitive join in the same conjunctive block, or a type check that is statically known to be true, such as the JAVA_TYPE<**byte**[]>(?ba) in our earlier example), or unification of joins (e.g., simplifying LOOKUP(m, k, _) and LOOKUP(m, k, v) to just LOOKUP(m, k, v)). There are also advanced optimizations for flattening and merging nested queries. For a good example, consider the query:

```
query(Map.contains(key) == newset):
  newset == query(Set.contains(value)): array[value] == key;
```

This query inverts the mapping of an array, producing a map where all the array values become keys for the sets of all array indices where each value appears. In PQIL, this translates (after domain inference) into:

```
Reduce[MAP(?key, ?newset):!result]: {
    ARRAY_LOOKUP_Object(?array; _, !key);
    Reduce[SET(?value):!newset]: {
        ARRAY_LOOKUP_Object(?array; !value, ?key);
    }
}
```

which is correct but inefficient: we iterate over all entries in array to bind key, then in the inner reduction iterate over array *again* to find all the values that map to key.

PQIL flattens the nested queries by allowing *nested reductors* and an accompanying optimization. Nested reductors are only usable in the 'value' field of maps and default maps, where they provide a more compact notation for reductions such as the above:

```
Reduce[MAP(?key, SET(?value)):!result]: { ARRAY_LOOKUP_Object(?array; !value, !key); }
```

Note that the variable ?newset has completely vanished from the query. Furthermore, the query (rewritten in this form) can be executed in a single traversal over array, reducing the iteration time from quadratic to linear.

The condition for this optimization to apply is that we have a reduction of the form

```
Reduce[MAP(?key, ?value):!result1] { ...before... Reduce[R:!value]: body ...after... }
```

such that *before* and *after* do not reference value. If this is the case, we rewrite to

```
Reduce[MAP(?key, R):!result1] { ...before... body ...after... }
```

4 Evaluation

We evaluate the efficiency of our prototype PQL implementation by applying it to a number of tasks from prior literature:

- *bonus*, the task of computing the salary bonuses of employees. This is a well-known example from the databases literature, employed, e.g., in Yang et al.'s map-reduce-merge work [17].

  In this task we are given a set of employees such that each employee has an associated department and a set of accumulated bonuses. We compute a map from each employee to the total bonus, modified by a departmental modifier factor.
- *threegrep*, the example discussed in sections 2.2 and 3, of finding all strings (in a set of 100-character strings) that contain the substring "012" [6].
- *webgraph*, a task defined by Yang et al. [17], in which we are given a set of documents and links between them: each document has a set of out-link objects, which identify the origin document and the document it points to. The task here is to identify the set of all documents that point to themselves via one point of indirection.
- *wordcount* is the task of computing the absolute numbers of occurrences of words in documents. We assume that the words have already been tokenized, stemmed etc., and are matched to a unique integer word ID. The result of this computation is a map from word IDs to the number of times they occur in a collection of documents.

We implemented each of the above tasks in several different ways:

- *pql*: As a PQL/Java query.
- *manual*: As a single-threaded Java method.
- *manual-para*: As multi-threaded, hand-optimized Java code.
- *sql*: As an SQL database query together with SQL database table configurations, both for Postgres and MySQL.
- *hadoop*: As a map-reduction running on the Hadoop framework.

SQL and Hadoop are not the most natural points of comparison technology-wise, but in conceptual terms they are the most closely related systems on Java that we are aware of.

In all of our implementations, we made sure to use the same container classes for results and intermediate computations, so as to not bias the evaluation results in that respect. We also used the same sources of data:

- *bonus* uses a set of employees and an array of departments. Bonuses are stored as a set as part of the employee object. *bonus* also stores an array of employees, indexed by employee number, which we use for the SQL and Hadoop implementations as well as for the manually parallelized version.
- *threegrep* uses an array of byte arrays to store the strings we are looking for.
- *webgraph* uses a set of document objects, each of which stores a set of link objects that contain references to the link target object. Document objects have unique IDs and are stored in an array, which we use for communicating with SQL and Hadoop, and for our manually parallelized implementation.
- *wordcount* uses the same representation as *webgraph* for documents, with each document containing an array of integers representing the words in the document body, in sequence.

Our manual and manually parallelized implementations were mostly straightforward, except as we note below. The manual implementations and the PQL implementations were the only ones that consistently used the 'canonical' data representation (i.e., sets for *bonus*, *webgraph*, and *wordcount*). All other implementations had to rely on auxiliary arrays/tables either for communication or for optimization (manually parallelized code). As a result, the SQL implementations shown are often simpler, but only because a single relation combines information from multiple Java data structures. The SQL version becomes significantly more complex if the translation code from Java is included. In contrast, the PQL implementation works directly on the Java structures, provides static type checking, and, arguably, is a better syntactic fit for everyday parallel tasks.

4.1 Benchmark Implementation Details

To understand the details of our benchmark implementations, it is helpful to recall some particulars of the systems we compare to. Hadoop processes all data as ⟨*key*, *value*⟩ pairs. It first goes through a map phase, which maps ⟨k,v⟩ to different ⟨k2,v2⟩, then aggregates all v2 for the same k2 in the reduce phase, which produces ⟨kOut,vOut⟩ pairs.

SQL databases do not expose direct access to their database internals, but instead use the JDBC interface for database-to-Java connectivity. We made an effort to consistently employ best-practice idioms, such as always using batch updates for setting up and prepared statements for updates and queries.

Bonus. The PQL query for *bonus* is:

```
result = query(Map.get(employee) == double bonus):
           employees.contains(employee) &&
           bonus == employee.dept.bonus_factor *
                 (reduce(sumDouble) v:
                   exists Bonus b: employee.bonusSet.contains(b) && v == b.bonus_base);
```

In the SQL implementation, we use three tables (employees, bonuses, departments) with suitable SQL types. The main SQL query is:

```
SELECT employees.eid, SUM(bonuses.base * factor)
  FROM employees
       JOIN departments ON departments.did = employees.dept
       JOIN bonuses ON bonuses.eid = employees.eid
  GROUP BY employees.eid
```

(The SQL code shown here and later intends to illuminate the main component of the benchmark in terms of performance. It is not representative for conciseness comparisons, since it omits large amounts of scaffolding code to convert the data from and to Java structures.)

For the Hadoop implementation, we make the department table available and transmit employees via employee ID, followed by department ID, followed by a sequence of bonus values. (All these form the map key, with a constant value—we encode the set of the other implementations into a Hadoop map.) The mapper sums up bonus values, looks up the department and multiplies. The reducer merely aggregates.

In our manually parallelized version of this benchmark we pre-allocated a result hash map to a sufficient size to avoid having to resize the map. Furthermore, we arranged for the employee hash method to map each key to a unique bucket in the table. This allowed us to have our parallel threads write to the same result table without contention.

Threegrep. We have already seen the PQL query for *threegrep* in Section 2.2. For the SQL implementation, data are stored in single table, as string ID (for communicating with Java heap) and char string. The core query is (for Postgres, with a slight difference for MySQL):

```
SELECT id FROM data WHERE (POSITION ("012" IN BODY) > 0)
```

This is a particularly friendly benchmark for SQL implementations. Once the data setup is complete, the above query is quite simple, with explicit support in the language for substring matching.

The Hadoop implementation uses an input map with key-value pairs of the form ⟨string-id:int, string:byte string⟩. The mapper outputs IDs of matching strings, and the aggregator does a straightforward aggregation.

In our manually parallelized implementation we used a synchronized result table shared among the worker threads.

Webgraph. The PQL query for *webgraph* is:

```
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;
```

benchmark	size (# objects)	Lines of Java code				
		manual	manual-parallel	Hadoop	SQL	PQL
bonus	2,360,000	9	50	130	48	8
threegrep	800,000	9	46	60	21	6
webgraph	92,000,000	13	50	105	39	4
wordcount	92,000,000	8	98	93	38	4

Fig. 2. Summary of our experimental setup, including heap size (approximate, due to randomization) and non-comment non-whitespace lines of Java code, excluding syntactic overhead from our benchmarking and import declarations, but including any encoding or decoding overhead required by the framework

The SQL implementation consists of three tables: webdocs (with IDs), links (source ID, target ID, link ID), and words (owner-webdoc-id, offset-in-doc, word-id). (The words table is used for the next benchmark, *wordcount*.) The SQL query is:

```
SELECT links.source FROM links
  JOIN links as links2
    ON links.destination = links2.source AND links2.destination = links.source;
```

For Hadoop, we provide tuples ⟨document-id, array-of-referenced-documents⟩ to the mapper, i.e., we flatten the link set to become part of the document during preprocessing. As output, the mapper produces one pair of each for ⟨src-doc, target-doc⟩ as well as ⟨target-doc, −1− src-doc⟩. (The use of negative 'id's serves to encode inlinks and outlinks differently.) The reducer aggregates all data for the same document and stores everything in a hashmap. If the reducer encounters both a link to 'd' and a link to '−1− d', it knows that we have a circle through 'd' and emits the current document.

Wordcount. The PQL query for *wordcount* is:

```
result = query(Map.get(int word_id) == int idf default 0):
           idf == reduce(sumInt) one over doc:
             one == 1 && documents.contains(doc) && exists i: doc.words[i] == word_id;
```

The SQL setup is the same as in *webgraph* but only uses the words table in a simple query:

```
SELECT word, COUNT(docid) FROM words GROUP BY word
```

The above illustrates our earlier point about PQL using the canonical representations of data, while the SQL and Hadoop implementations can use auxiliary data structures. The words table combines both the documents and the words structures in the PQL implementation. This representational simplification permits a very concise SQL query, but comes at a cost in run-time and code size during setup.

The Hadoop implementation is straightforward, with the mapper input in the form ⟨doc-id, words-as-int-array⟩, mapper output as ⟨word-id, counts-of-word-in-doc⟩, and reducer output ⟨word-id, aggregate-counts-of-word⟩. Figure 2 summarizes our experimental setup. We note that PQL is significantly more compact than any of the parallel alternatives, even though we often broke up our query expressions generously across multiple lines of code, for readability. We did not do the same for SQL: since SQL statements are encoded as strings in Java, formatting them is inconvenient.

4.2 Configuration

We ran our experiments on a Sun SPARC64 (Sparc v9) Enterprise-T5120 system, with 8 cores at 8 SMT threads each, and a dual 6-core Intel Xeon X5650 machine. As runtime we used the native Java 1.7.0-01 on Sun and 1.6.0_26-b03 on Intel, configured to pre-allocate 2 GiB of RAM. For our Hadoop experiments we used the most recent release, Hadoop 0.20.205.0. Our SQL experiments we conducted both on PostgreSQL 8.3.1 and MySQL 5.1.46. The databases were set up to run locally, with no special tuning parameters except as mentioned in the previous section.

PQL Compiler and Runtime. We configured our PQL runtime for its default execution model. In this setup, our runtime executes the body of a reduction in parallel, with each evaluation thread processing a slice of the index space the reduction body has to iterate over. Once a thread has finished its own slice of the 'embarrassingly parallel' initial computation, it follows a pre-assigned merge procedure in which it waits for and merges the results of other threads, in a binary tree fashion: on every round, each thread synchronizes with one other thread to merge their results, and one of them goes on to the next round for further merging. This guarantees that there are precisely n merge steps. We use no further synchronization.

We also enabled all optimizations in our compiler, specifically redundant join elimination, access path selection, and the nested join optimization and translation to Java code (rather than interpretation).

4.3 Measurement Results

Figures 3 and 4 (for the Intel and Sun architectures, respectively) summarize the running times of our PQL implementation on the four benchmarks. The run-times vary from around 100 ms (threegrep on Intel) to roughly half a minute (webgraph on SPARC). Our graphs only show curves for the PQL, manual (single-threaded) and manual-para (hand-optimized multithreaded) versions. The Hadoop and SQL curves are excluded since they skew the results and interfere with their visualization. Hadoop and SQL consistently suffer from low overall performance due to the cost of transferring data back and forth between different heaps and converting it between representations. For reference, we show, in log-scale, the performance of all six implementations for a single-threaded run (i.e., running on a single core, even for the parallel versions) of *threegrep*, in Figure 5. For this benchmark we reduced the size of *threegrep*'s data to $\frac{1}{10}$th.

It is clear that the SQL and Hadoop implementations do not perform at nearly the same level as techniques for running in the same memory space. For native heap execution, setup and result transfer time are negligible. For Hadoop, setup time is small, since we translate values to efficient binary representations, but for both SQL implementations it is prohibitive, most likely due to the necessity of (de)serializing to and from text. Without this overhead MySQL comes within an order of magnitude (but still several times slower) of our native implementations. We chose *threegrep* as a representative benchmark since it is comparatively small and reports only a few dozen results, permitting negligible result transfer cost (which we do not report separately). Our measurements for larger benchmarks are comparable or worse for SQL, though Hadoop

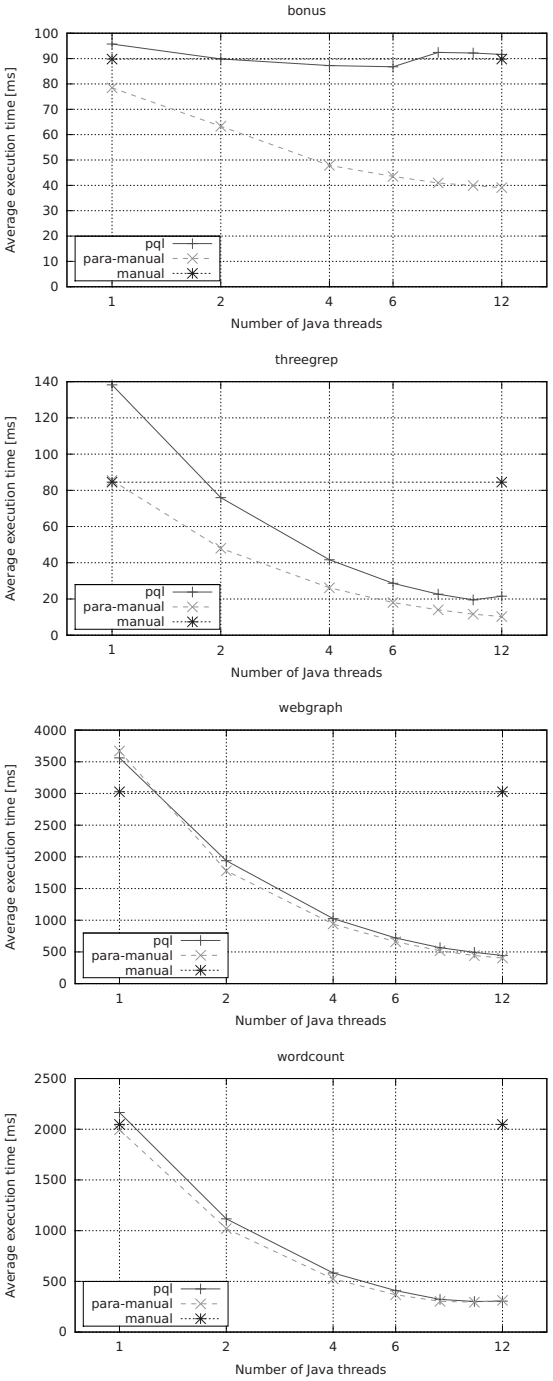

Fig. 3. Results for Intel architecture

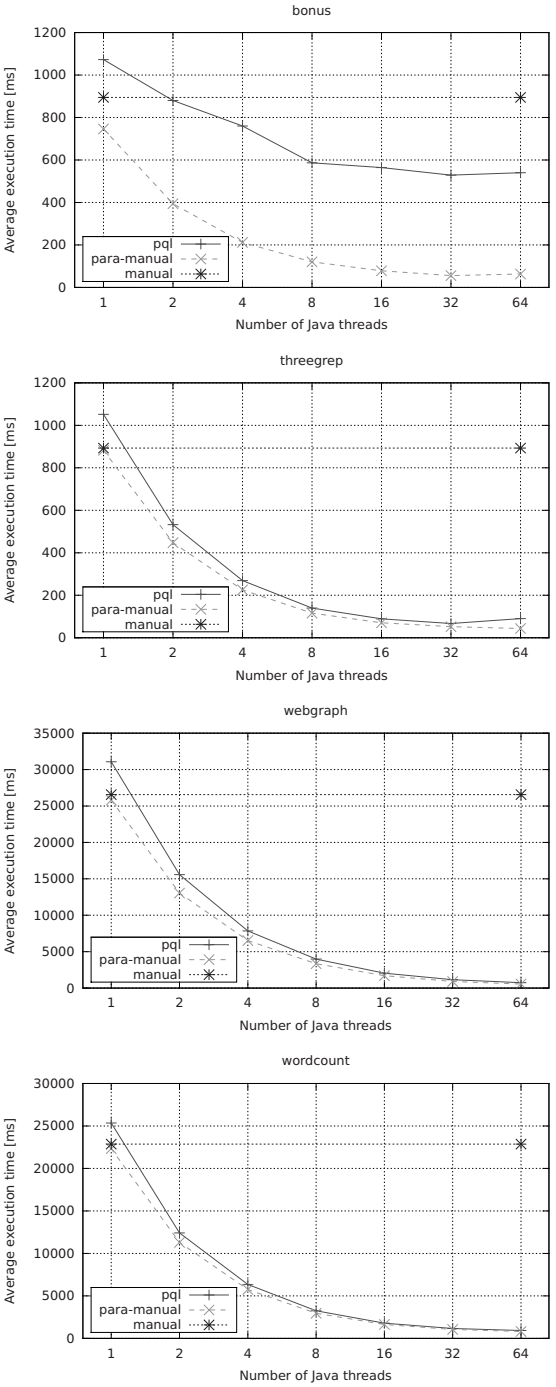

Fig. 4. Results for Sun architecture

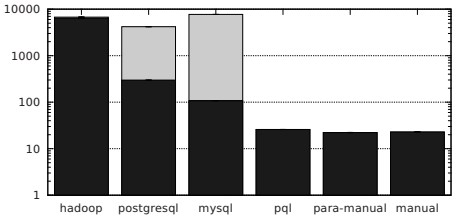

Fig. 5. Results (runtime in ms), in log-scale for all implementations, single-threaded (i.e., on one core only) run of *threegrep* on the Sun architecture. Lighter segments in the bars show setup overheads (i.e., initialization time)—these are large enough to be discernible only for SQL implementations. The figure contains error bars that are non-discernible at this scale

improves (relatively speaking), e.g., Hadoop on regular *threegrep* is "only" 25× slower than our baseline, compared to 84× at reduced benchmark size. Measurements with multithreaded Hadoop show slight improvement, but not enough to be discernible at the scale of Figure 5.

On the more interesting topic of PQL performance compared to manually tuned Java code, we see that the PQL implementation scales in roughly the same patterns as manual code, and nearly matches manual code performance. For *webgraph*, *wordcount*, and *threegrep*, the performance of PQL is strong on both architectures. The remaining benchmark, *bonus*, scales less ideally. The culprit is contention and the high cost of merging results, which dominates the cost of producing the per-processor results. As we described earlier, the manual implementation of *bonus* exploits knowledge about the amount of results it will produce, to pre-allocate a table of the right size and avoid all locking. Such powerful manual optimizations are hard for compilers to reason about or devise. This low-level optimization is particularly important for *bonus* because of its otherwise simple computation. In the PQL implementation, the merge component becomes comparatively bigger than the embarrassingly parallel computation component. At 64 threads, the runtime overhead of a merge is 5.64 for *bonus* (i.e., merge cost is over 5 times as high as computation cost)! (Comparatively, this overhead is 0.01 for *webgraph* and *wordcount*.) The merge overhead thus overrides much of the benefit of parallelization. On Intel, the effect is more pronounced, especially once we exceed 6 threads and start using simultaneous multi-threading.

Wordcount depends critically on our nested join optimization (Section 3): This optimization merges the blocks of joins of inner and outer reduction and thus gives us greater freedom during access path selection. In practice, this allows us to traverse over all documents as the outermost loop, allowing each worker thread to analyze a subset of documents. Without the inner reduction optimization, the only viable outermost loop would be over all array indices in all documents, which means that each thread would have to touch (a slice of) each document. We validated that this rejected access path would parallelize poorly.

For completeness, we list PQL and manually parallelized overhead compared to the baseline, together with the speedups (inverse overhead) observed for PQL, in Figures 6 and 7.

	Overhead over manual		PQL speedup						
benchmark	**para-manual**	**PQL/1**	**1**	**2**	**4**	**6**	**8**	**10**	**12**
bonus	$1.14 + 0.00\sigma$	$0.94 + 0.00\sigma$	0.94	1.00	1.03	1.03	0.97	0.97	0.98
threegrep	$0.99 + 0.00\sigma$	$0.61 + 0.00\sigma$	0.61	1.11	2.03	2.95	3.73	4.33	3.93
webgraph	$0.82 + 0.00\sigma$	$0.85 + 0.00\sigma$	0.85	1.56	2.95	4.20	5.33	6.12	6.83
idf	$1.03 + 0.00\sigma$	$0.95 + 0.00\sigma$	0.95	1.83	3.51	5.00	6.35	6.78	6.72

Fig. 6. Overhead and speedup measurements on Intel

	Overhead over manual		PQL speedup						
benchmark	**para-manual**	**PQL/1**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
bonus	$0.83 + 0.00\sigma$	$1.20 + 0.00\sigma$	0.83	1.02	1.18	1.52	1.58	1.69	1.66
threegrep	$0.99 + 0.00\sigma$	$1.18 + 0.00\sigma$	0.85	1.68	3.32	6.39	10.01	13.27	9.91
webgraph	$0.97 + 0.00\sigma$	$1.17 + 0.00\sigma$	0.85	1.71	3.39	6.68	12.98	23.16	35.61
idf	$0.98 + 0.01\sigma$	$1.11 + 0.00\sigma$	0.90	1.84	3.60	7.04	12.78	19.63	24.73

Fig. 7. Overhead and speedup measurements on Sun

Overall, the experiments show how *casual in-memory tasks can benefit from PQL, making seamless declarative parallel processing possible* in the middle of a Java application. Achieving the observed level of performance is the result of significant optimization in the PQL back-end—our original unoptimized implementation (also exploiting parallelism) was more than 10 times slower.

4.4 Discussion

In practice, applications that rely on databases usually store data separately from the Java heap. Doing so in our context would have eliminated the setup cost (though not the query transfer cost). However storing data in databases comes at the price of a semantic gap between Java and the data representation: we cannot add methods to database tables, refactor them in a Java IDE or write unit tests for them easily. The semantic gap extends to the language. We found SQL and PQL to be the languages with the most concise ways to express the computations we were interested in. However, SQL operates on database tables, not sets, maps, objects, and arrays; we thus found it to be an imperfect match for the queries we wished to express. As we saw in Table 2, the programmatic cost of bridging this semantic gap can be considerable. PQL/Java avoids the gap altogether, making declarative parallel programming easy for everyday tasks.

We found (not unexpectedly, but to a larger degree than expected) that Hadoop is not designed for data processing at such a (comparatively) fine-grained scale, i.e., for data that fits into a single computer's memory. For such tasks, we found Hadoop's overhead to be prohibitive. The amount of implementation work needed to communicate with Hadoop efficiently was significantly greater than the amount needed for SQL, since not all queries fit obviously into a map-reduce framework (esp. *webgraph*'s).

For a fair comparison, we should note that Hadoop and SQL databases provide additional features, specifically persistence layers, that are beyond the scope of PQL. However, our experiments suggest that programmers who do not need such persistence and

are only interested in efficient, parallelizable queries that fit within the Java heap have much more to gain from PQL than from (mainstream, unspecialized) SQL database engines or Hadoop.

5 Related Work

It is virtually impossible to be comprehensive when describing related work in parallel languages. There have been numerous and quite diverse approaches, spanning multiple decades. Compared to all of them, the distinguishing feature of our work is that it promotes purely declarative extensions for parallelism, yet keeps the close integration between the declarative sub-language and the sequential host language, with both operating on the same data.

In terms of programming model, the PQL/Java approach can be viewed as map-reduce-on-steroids. Map-reduce computations have a simple, fixed structure that is an easy-to-express special case of our declarative language. PQL/Java generalizes this to offer a full logic-based language in which complex program flows can be expressed and automatically parallelized/optimized. For instance, instead of a plain map-reduce-like structure, an application in our system can have a forall-exists-forall structure, examining combinations of existing data structures and not just mapping over a single one. This need has already been identified in the map-reduce domain. For instance, Google recently introduced the FlumeJava library [4], which supports "a pipeline of MapReduces". In terms of control-flow structures, this is again a special case of our declarative language: the parallel program structures expressible in FlumeJava can also be expressed in PQL/Java. Furthermore, our approach has a much higher-level nature, as it allows aggressive automatic optimization—a direction that FlumeJava begins to pursue with fusion-like parallel loop optimizations, but cannot exploit to nearly the same extent. Of course, directly comparing to specific map-reduce facilities is not appropriate, because the focus of our work is quite different: PQL/Java only targets shared-memory parallelism,[5] while map-reduce libraries are distinguished by their support for distributed, fault-tolerant parallel computations.

Relational databases also have the declarative flavor of the PQL/Java approach, and there is intense research and practical interest in integrating support for relational queries into mainstream programming languages. Microsoft's LINQ and its parallel version, PLINQ [8], are some of the best known such facilities. We already discussed how the design of PLINQ is explicitly not as declarative as that of PQL/Java. Furthermore, we believe that SQL-like syntax is a mismatch for general purpose parallelism: expressing an arbitrary computation with SQL operators such as select, project, join, and difference is awkward. In contrast, we offer a language that has a much more explicit looping structure (forall and exists loops), and an optimizer that leverages the accumulated knowledge from database optimizers, while also understanding the structure of first-order logic sentences. Finally, mainstream relational database engines, such as

[5] The language is designed with the prospect of distributed execution in mind, for future incarnations. The current implementation is for shared-memory machines, however. Other parallel languages—e.g., Fortress [1]—follow the same pattern of starting from shared-memory but designing with an eye for distribution as well.

MySQL and Postgres, do not offer parallelization of a single query, although they support parallel execution of separate queries. This is another example of how applying relational techniques to the usual objects of a Java heap decisively changes the language implementation tradeoffs: Intra-query parallelization makes sense for read-only in-memory data, but less so for traditional transactions in a real database.

In terms of language support for parallelism, there is a multitude of designs that are impossible to cover exhaustively, but follow lines quite distinct from our work. These designs can be as simple as libraries for task-parallelism (e.g., offering a "parallel-for" primitive [12]) and as complex as entire languages for matrix computations, media processing, stream processing, etc. [15,3,10]. Compared to the former, our approach aims to be higher-level, due to the declarativeness of parallel computation. That is, task-parallel libraries only hide the specific mechanisms for parallelism but do not otherwise help address the inherent difficulty of parallel programming. The user is still burdened with structuring the parallel program and little optimization takes place automatically, unlike in PQL/Java. Compared to domain-specific mechanisms for parallelism, our approach is explicitly unifying, with a general declarative language for a substantial subset of all parallel programming tasks.

Finally, PQL/Java is conceptually related to languages that emphasize concurrency and avoid imperative features. It is not a new observation that declarativeness is a good match for parallelism. For instance, the "Declarative Aspects of Multicore Programming" (DAMP) workshop has been held since 2006 and has hosted the presentation of several approaches relating to declarative support for concurrency. Past approaches, however, are typically less general or less declarative than our pure logic-based approach—we offer the first approach that is completely declarative (based on first-order logic, which is truly a specification language, without order dependencies and side-effects), general (can express in a single language the parallel elements of programs from different domains), and unified with a sequential language in a way directly inspired by complexity theory. For a representative comparison, Erlang [2] is a celebrated success story of declarative languages in parallel programming. Nevertheless, Erlang is a Turing-complete and not purely declarative language. (E.g., it is not the case that clauses can be freely reordered without affecting program meaning.) This means that, although Erlang program components communicate asynchronously and, thus, can be easily run in parallel, the responsibility for structuring the program is left with the programmer. Similar comments apply to most high-level languages explicitly designed with parallelism in mind, such as Fortress [1] and X10 [5]. (As with map-reduce mechanisms, however, X10 explicitly targets the much harder problem of distributed execution, while we focus on shared-memory parallelism.) In contrast to such work, our declarative approach consists of specifying in queries what is to be done in parallel but not the exact parallel program structure. The terms of such queries can be freely reordered, factored, and aggressively optimized by the runtime system. Also, the runtime system is fully responsible for deciding what constitutes a task that gets assigned to a processor, unlike in Erlang, where this is dictated by program structure. In short, it is important that our declarative language is not by itself a full, Turing-complete language: the potential for automatic optimization and parallelization is much higher.

We consider this feature crucial for getting higher-level programmability and addressing the challenges of parallel programming.

6 Conclusions

We presented PQL/Java: an approach to parallel programming that employs a purely declarative sublanguage for parallelism, integrated with a mainstream language for sequential computation. PQL queries operate over regular Java data and get automatically optimized by exploiting the declarativeness of the specification. PQL is not a full programming language but it is well-suited for combining, filtering, and reducing large data structures, in a control flow that generalizes map-reduce patterns. Because PQL is a general logic, we expect its users to find innovative ways to express interesting computations, beyond the motivating examples of the original design, fulfilling the promise of a truly high-level, programmer-friendly parallel sublanguage.

Acknowledgments. We would like to thank the anonymous ECOOP reviewers for their feedback. This work was funded by the National Science Foundation under grants CCF-0917774, CCF-0934631, and CCF-1115448.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G., Tobin-Hochstadt, S.: The Fortress Language Specification. Technical report, Sun Microsystems (2008)
2. Armstrong, J.: A history of Erlang. In: HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages (2007)
3. Catanzaro, B., Fox, A., Keutzer, K., Patterson, D., Su, B.-Y., Snir, M., Olukotun, K., Hanrahan, P., Chafi, H.: Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. IEEE Micro 30(2), 41–55 (2010)
4. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: easy, efficient data-parallel pipelines. In: Programming Language Design and Implementation, PLDI (2010)
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2005)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Operating Systems Design & Implementation (OSDI) (2004)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
8. Duffy, J.: A query language for data parallel programming: invited talk. In: Declarative Aspects of Multicore Programming Workshop, DAMP (2007)
9. Duffy, J., Essey, E.: Parallel LINQ: Running queries on multi-core processors. MSDN Magazine (2007)
10. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ASPLOS-XII: Architectural Support for Programming Languages and Operating Systems (2006)
11. Immerman, N.: Descriptive Complexity. Springer (1998)

12. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2009)
13. Reichenbach, C., Immerman, N., Smaragdakis, Y., Aftandilian, E.E., Guyer, S.Z.: What can the GC compute efficiently? A language for heap assertions at GC time. In: Object Oriented Programming Systems, Languages, and Applications, OOPSLA (2010)
14. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: ACM SIGMOD Int. Conf. on Management of Data, pp. 23–34 (1979)
15. Snyder, L.: The design and development of ZPL. In: HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages (2007)
16. Willis, D., Pearce, D.J., Noble, J.: Efficient Object Querying for Java. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 28–49. Springer, Heidelberg (2006)
17. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: ACM SIGMOD Int. Conf. on Management of Data (2007)