
PQL: A Purely-Declarative Java Extension for Parallel Programming

Christoph Reichenbach¹, Yannis Smaragdakis²,
Neil Immerman³

1: Google (but soon: Goethe University Frankfurt)

2: University of Athens

3: University of Massachusetts, Amherst

WRITING PARALLEL PROGRAMS IS HARD

- locking
- races
- side effect order
- consistency models
- distributing computations
- . . .

EASIER PARALLELISM

Approach	Problems	User actions
map-reduce	emb. parallel + aggregation	split computation
fork-join	divide-and-conquer	(recursively) divide up problem
PLINQ	SQL-like, over containers	tag parallel steps
Pregel	graph algorithms	split into graph computations, -mutations

Frameworks for *manual* parallelisation

Casual parallelism: *fully automatic*

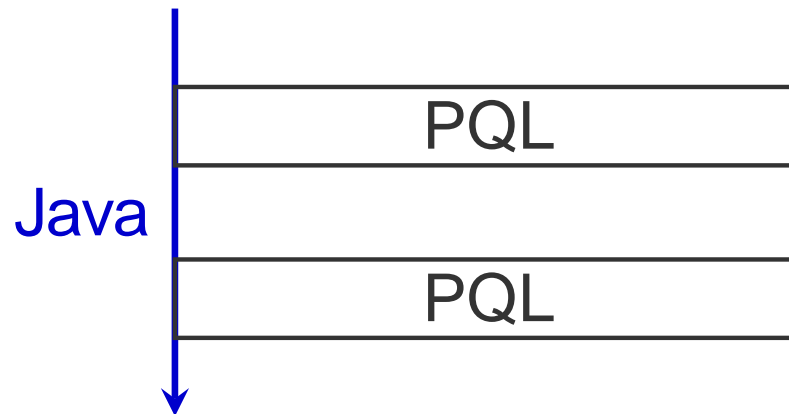
CASUAL PARALLELISM

- Pitfalls:
 - Side effects
 - Order dependency

Declarative language
Specify the '*what*', not the '*how*'

PQL/JAVA

- Declarative extension to Java:
Parallel Query Language
- Fully automatic parallelisation
- Processes and builds Java containers



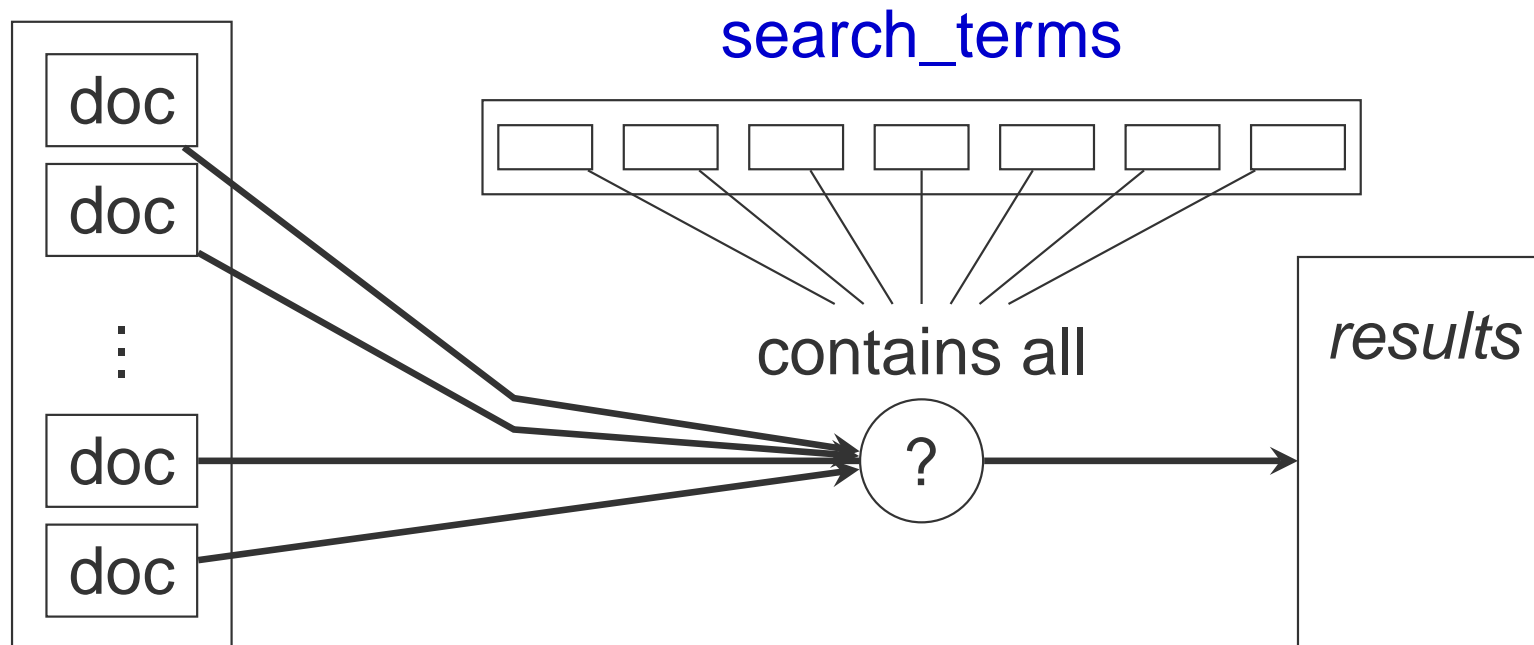
PQL/JAVA

- Declarative extension to Java:
Parallel Query Language
- Fully automatic parallelisation
- Processes and builds Java containers

Java for sequential code, PQL for parallel code

PQL EXAMPLE

DocRepository



query (Set.contains(doc)):

`DocRepository.getAll().contains(doc)`

`&& forall x: doc.contains(search_terms[x]);`

WHAT RICH LANGUAGE GIVES US CASUAL PARALLELISM?

- Embarrassingly parallel:
Executable in $O(1)$ with enough CPUs
- Result from *Descriptive Complexity*:
This language is precisely First-Order Logic^a

Using $O(n^3)$ cores may be a bit much...

^aif we assume a polynomial number of CPUs

MAKING FIRST-ORDER LOGIC MORE USEFUL

- *Assert or compute* results:
 - Finite set comprehension
 - SQL-style queries (minus aggregation, ordering)

x	$\exists y. \mathbf{a}[x] = \mathbf{b}[y]$
0	true
1	false
2	false
3	true
\vdots	\vdots

} representation:
 $\Rightarrow \{0, 3, \dots\}$

ADDING REDUCTION

reduce(add) x **over** i: x == a[i]

- log-parallel performance
- user-supplied reducers

PQL OVERVIEW

- +, −, . . . , ?:, ==, instanceof, &&, ||, −>
- forall , exists
- Java expressions as constants
- m[k], m.get(k), c.length, c.size(), s.contains(e)
- Container construction:
 - query (Set.contains(int x)): ...
 - query (Array[x] == float f): ...
 - query (Map.get(String s) == int i [default v]): ...
- reduce(sumInt) int x [over y]: ...

MORE PQL EXAMPLES

- Check sortedness of list

MORE PQL EXAMPLES

```
assert forall Node n: sorted_list.contains(n)
  -> n.prev.value <= n.value;
```

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering

MORE PQL EXAMPLES

- Check sortedness of list

Set<Item> intersection =

```
query (Set.get(Item element)):
    set0.contains(element)
    && set1.contains(element)
    && !element.is_dead;
```

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering
- Employee bonus table

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering

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

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering
- Employee bonus table
- Vector dot product

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering
- Employee bonus table

```
dot_product = reduce(add) x over y: x == a[y] * b[y];
```

MORE PQL EXAMPLES

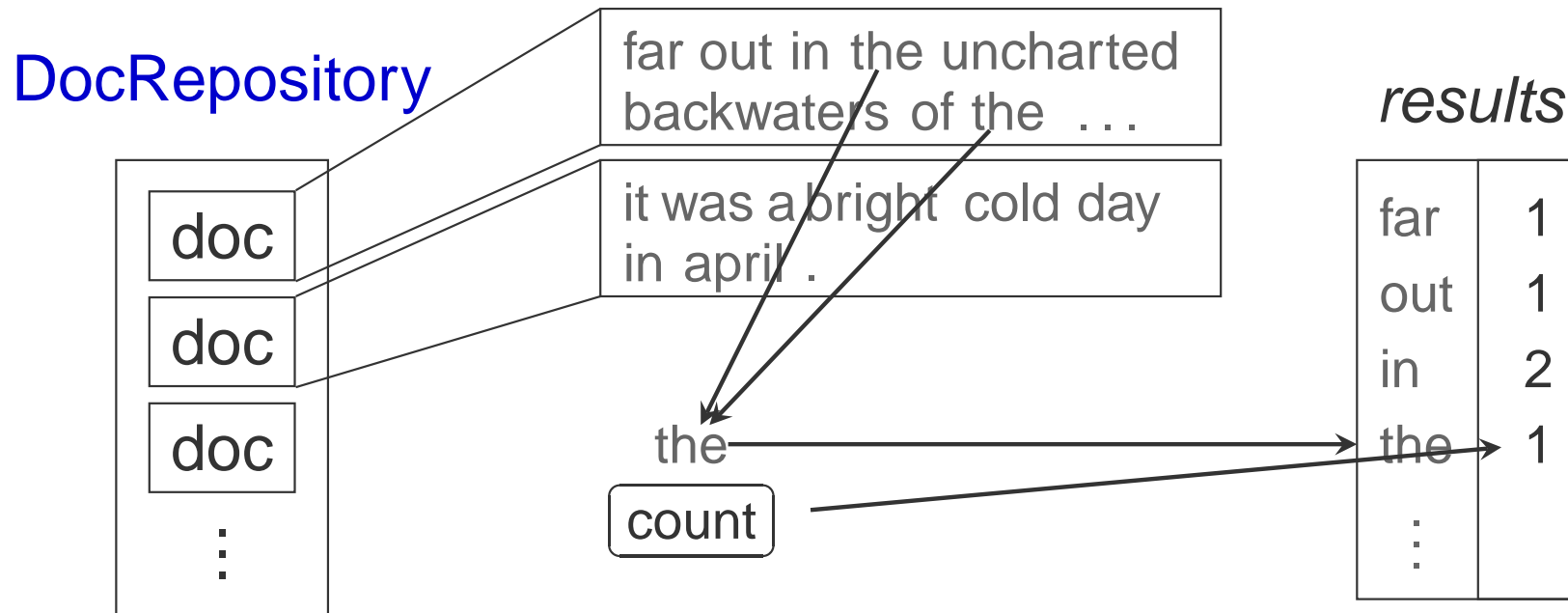
- Check sortedness of list
- Set intersection together with filtering
- Employee bonus table
- Vector dot product
- Invert map
- ...

MORE PQL EXAMPLES

- Check sortedness of list
- Set intersection together with filtering
- Employee bonus table
- Vector dot product

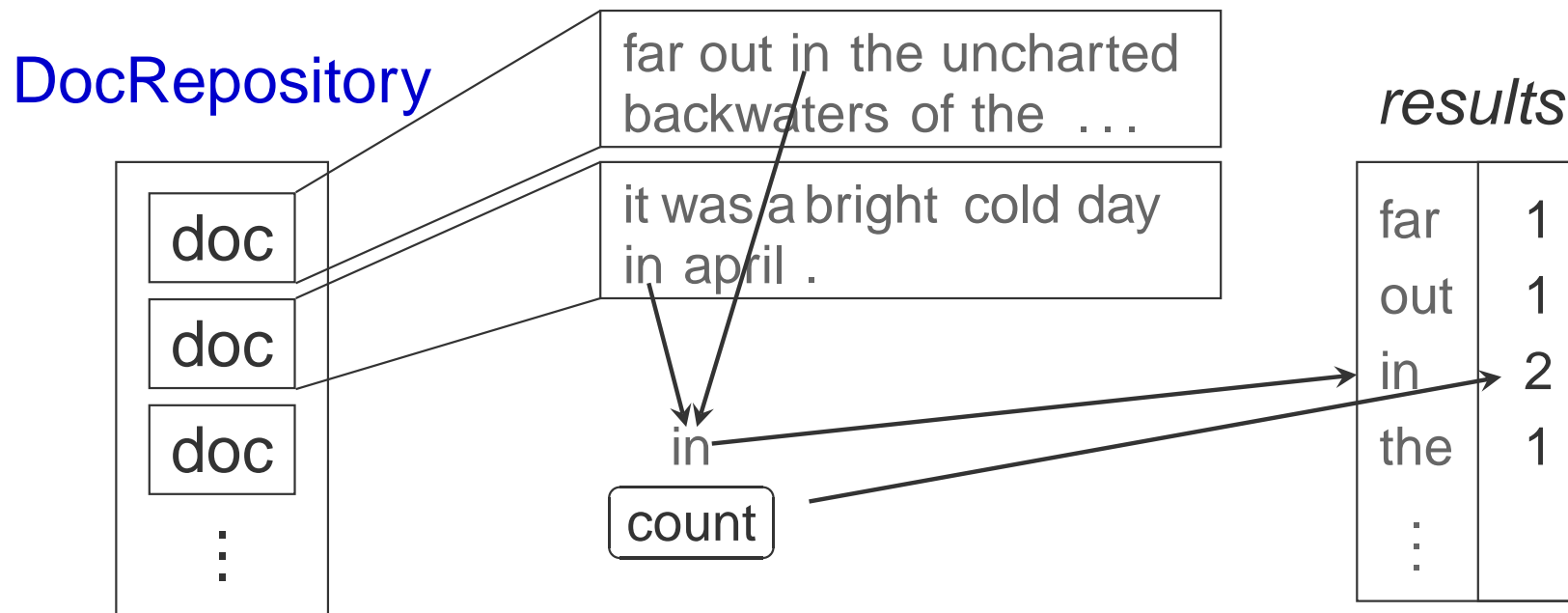
```
query (Map.find(value) == keyset default new PSet()):  
    keyset == query (Set.contains(key)):  
        m.get(key) == value;
```

REALISTIC PQL EXAMPLE



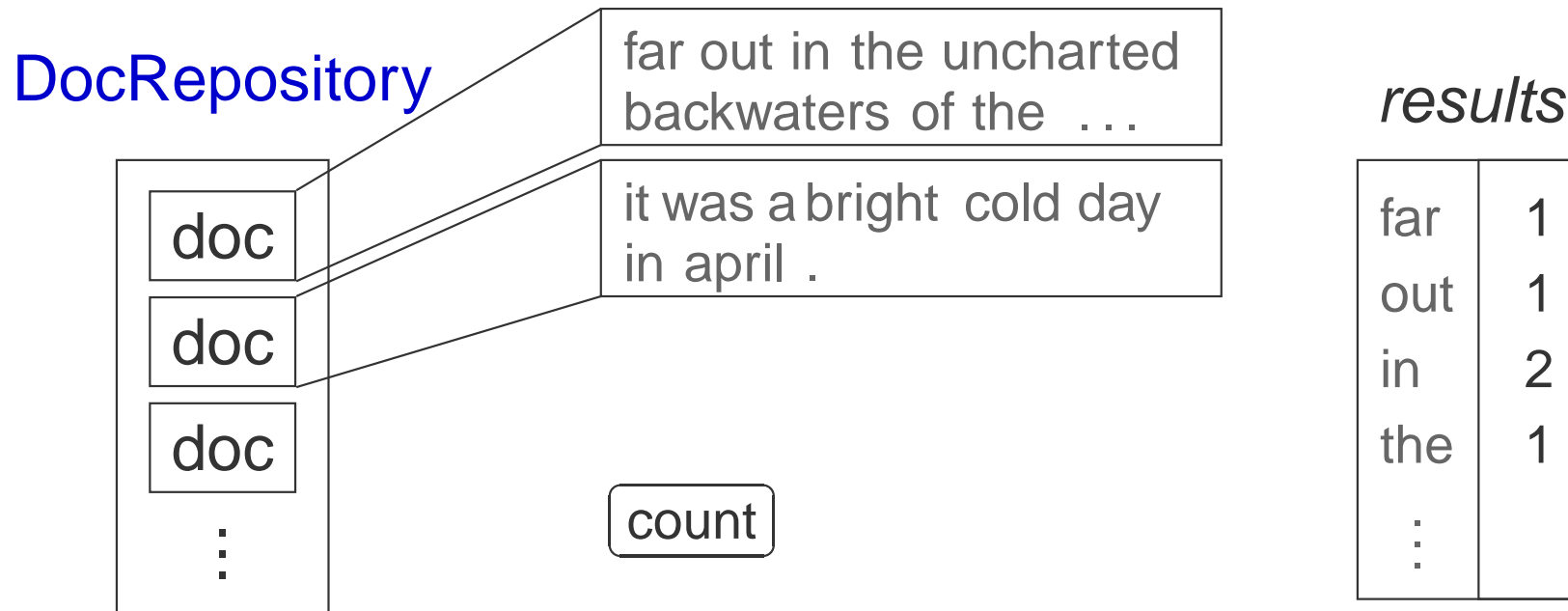
`DocRepository.getAll().contains(doc)`

REALISTIC PQL EXAMPLE



`DocRepository.getAll().contains(doc)`

REALISTIC PQL EXAMPLE



query (Map.get(int word_id) == int wcount **default** 0):

wcount == **reduce**(sum) **1** **over** doc:

DocRepository.getAll().contains(doc)

&& exists i: doc.words[i] == word_id;

IMPLEMENTATION

- Extension to javac 1.6:
 - PQL to relations
 - Access path selection / Query scheduling
 - Optimisation
 - Code generation
- Run-time library support:
 - parallel execution

EXAMPLE

reduce(max) **int** x : $a[x] > 0$

Int(x)

Gen. relational IL

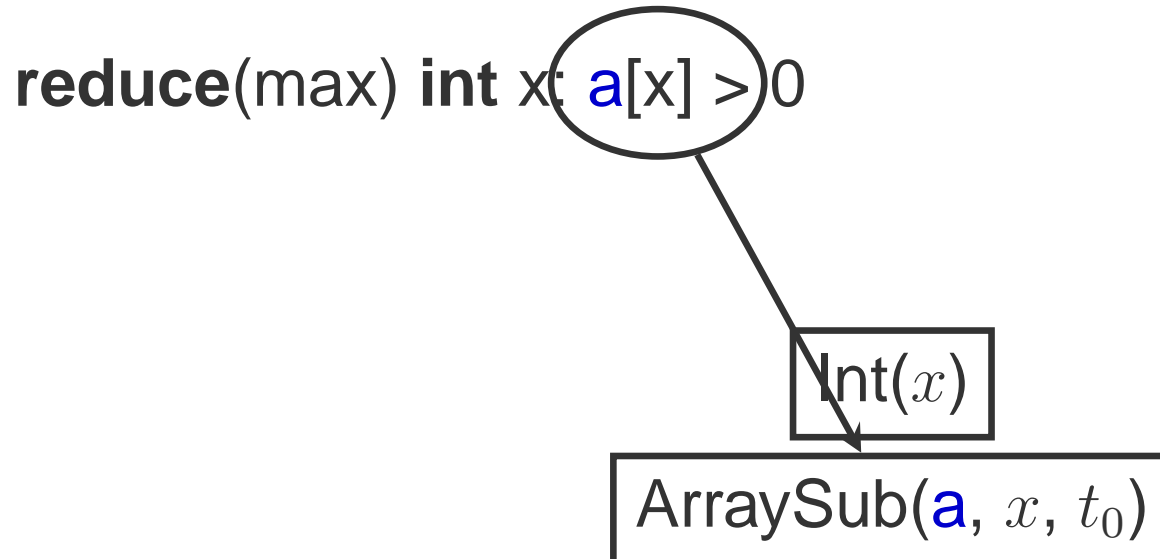
Query ordering

Optimisation

Code generation

Translation into relational IL

EXAMPLE



Gen. relational IL

Query ordering

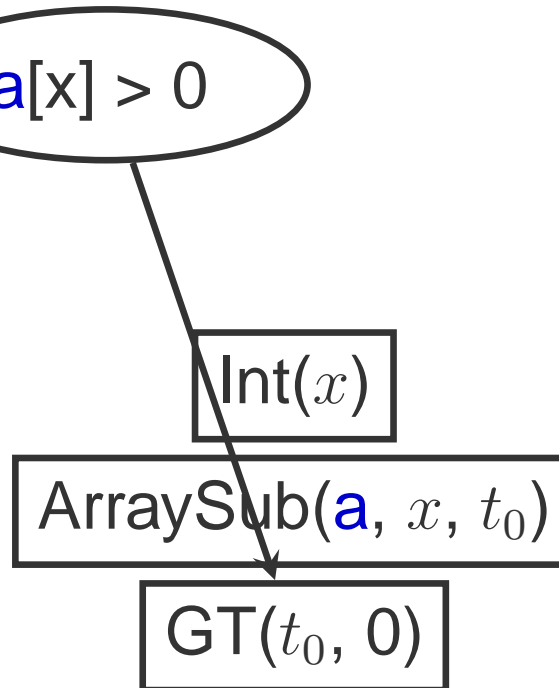
Optimisation

Code generation

Translation into relational IL

EXAMPLE

reduce(max) int $x: a[x] > 0$



Gen. relational IL

Query ordering

Optimisation

Code generation

Translation into relational IL

EXAMPLE

reduce(max) int x: a[x] > 0

Gen. relational IL

Query ordering

Optimisation

Code generation

Int(x)

ArraySub(a , x , t_0)

GT(t_0 , 0)

Unordered!

EXAMPLE

reduce(max) int x: $a[x] > 0$

Gen. relational IL

Query ordering

Optimisation

Code generation

$\text{Int}(x^w)$

$\text{ArraySub}(a^r, x^r, t_0^w)$

$\text{GT}(t_0^r, 0)$

Order #1: Must iterate over 2^{32} values!

EXAMPLE

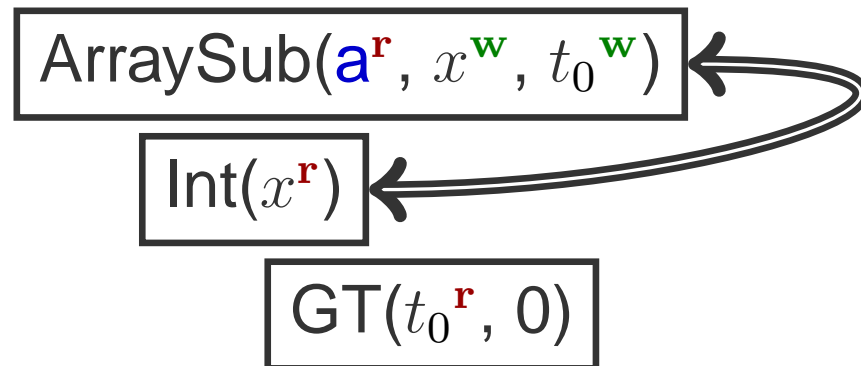
reduce(max) int x: a[x] > 0

Gen. relational IL

Query ordering

Optimisation

Code generation



Order #2: Iterate over a.length values

EXAMPLE

reduce(max) int x: a[x] > 0

Gen. relational IL

Query ordering

Optimisation

Code generation

ArraySub(a^r , x^w , t_0^w)

~~Int(x)~~

GT(t_0^r , 0)

EXAMPLE

Gen. relational IL

Query ordering

Optimisation

Code generation

```
for (x = 0; x < a.length; x++) {  
    t_0 = a[x];  
    if (t_0 > 0)  
        // signal success at x  
}
```

EXAMPLE

```
void runWorker(int start, int stop) {  
    for (x = start; x < stop; x++) {  
        t_0 = a[x];  
        if (t_0 > 0)  
            // signal success at x  
    }  
}
```

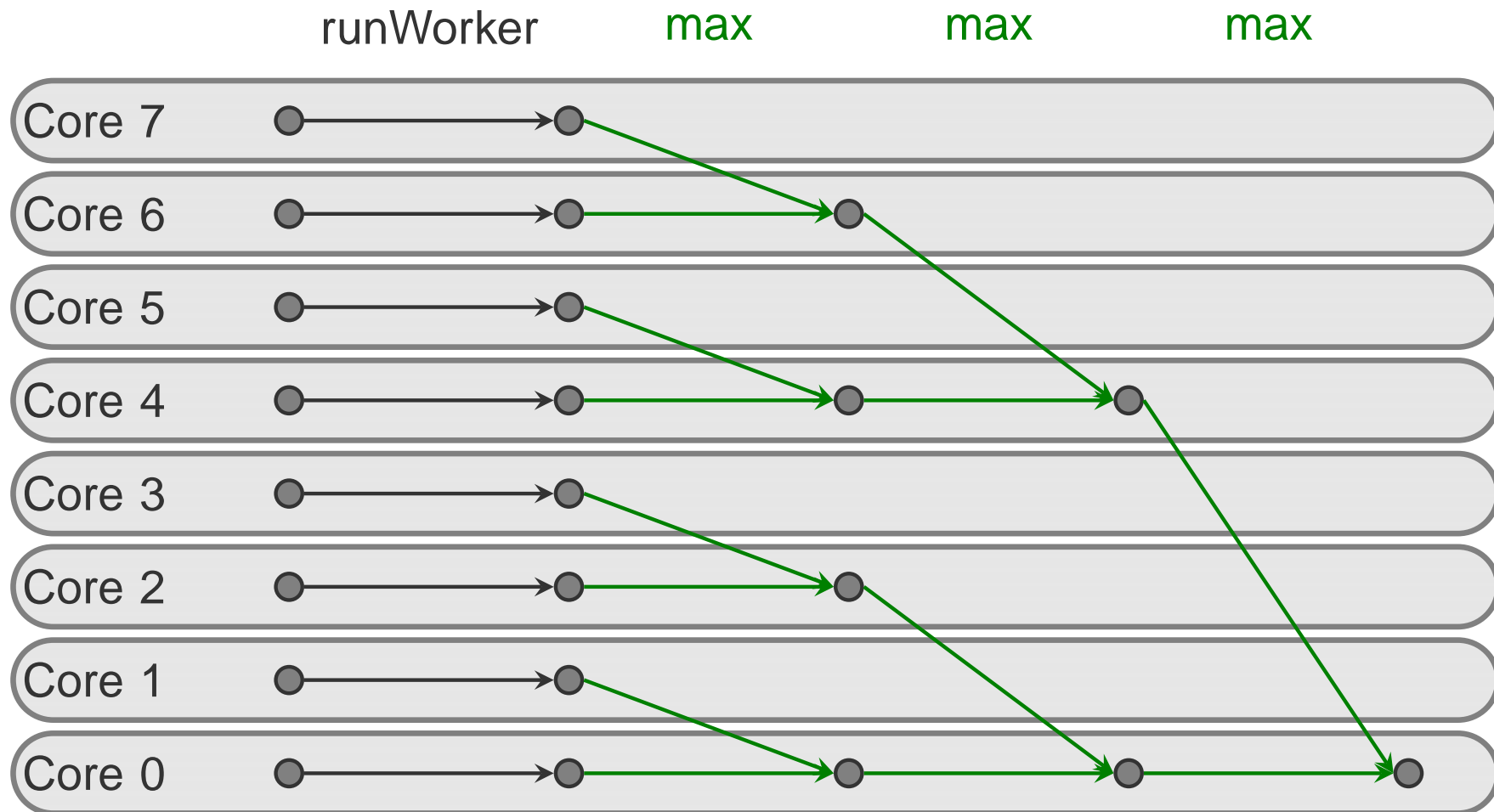
Gen. relational IL

Query ordering

Optimisation

Code generation

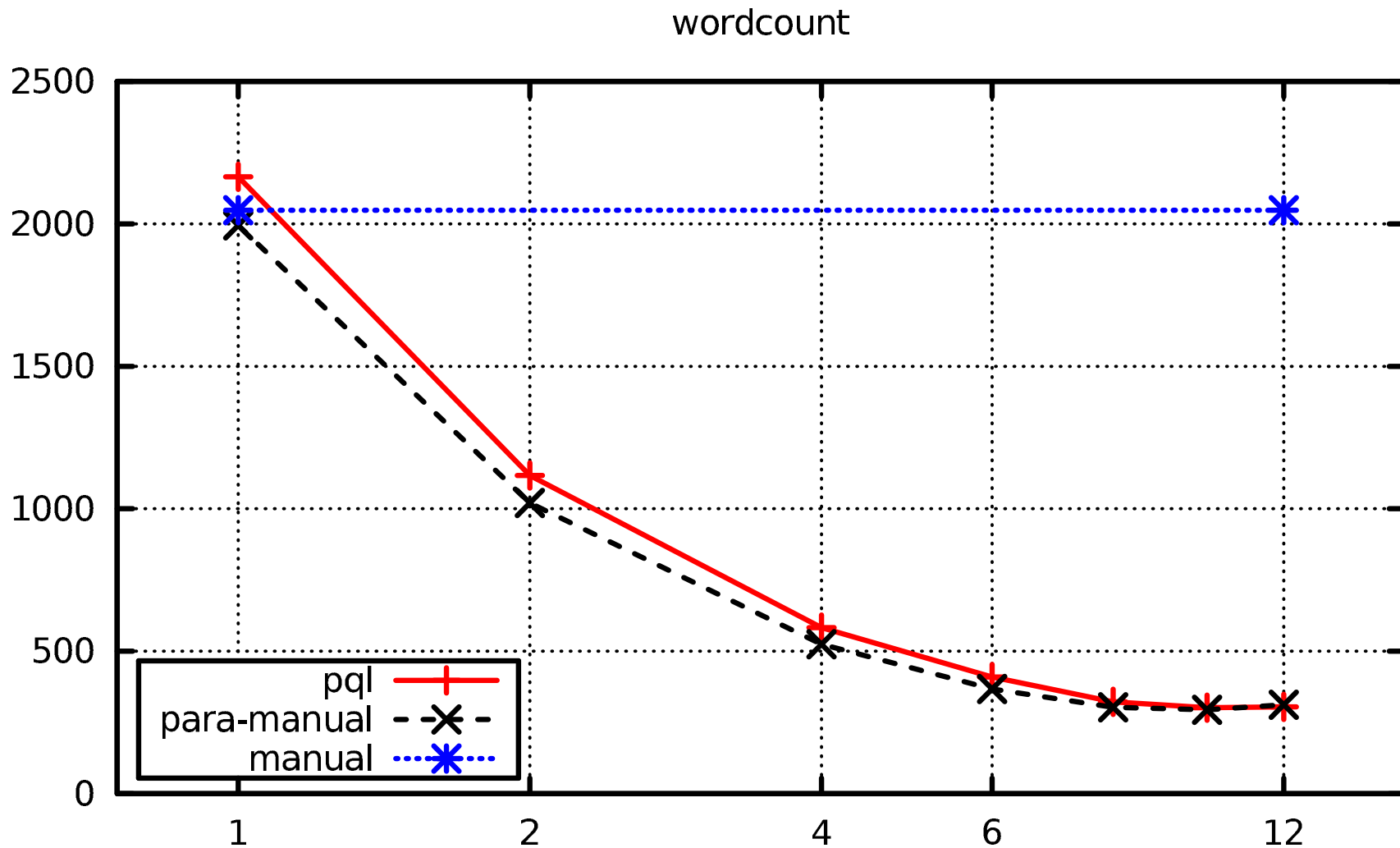
PARALLEL EXECUTION MODEL: TREE JOIN



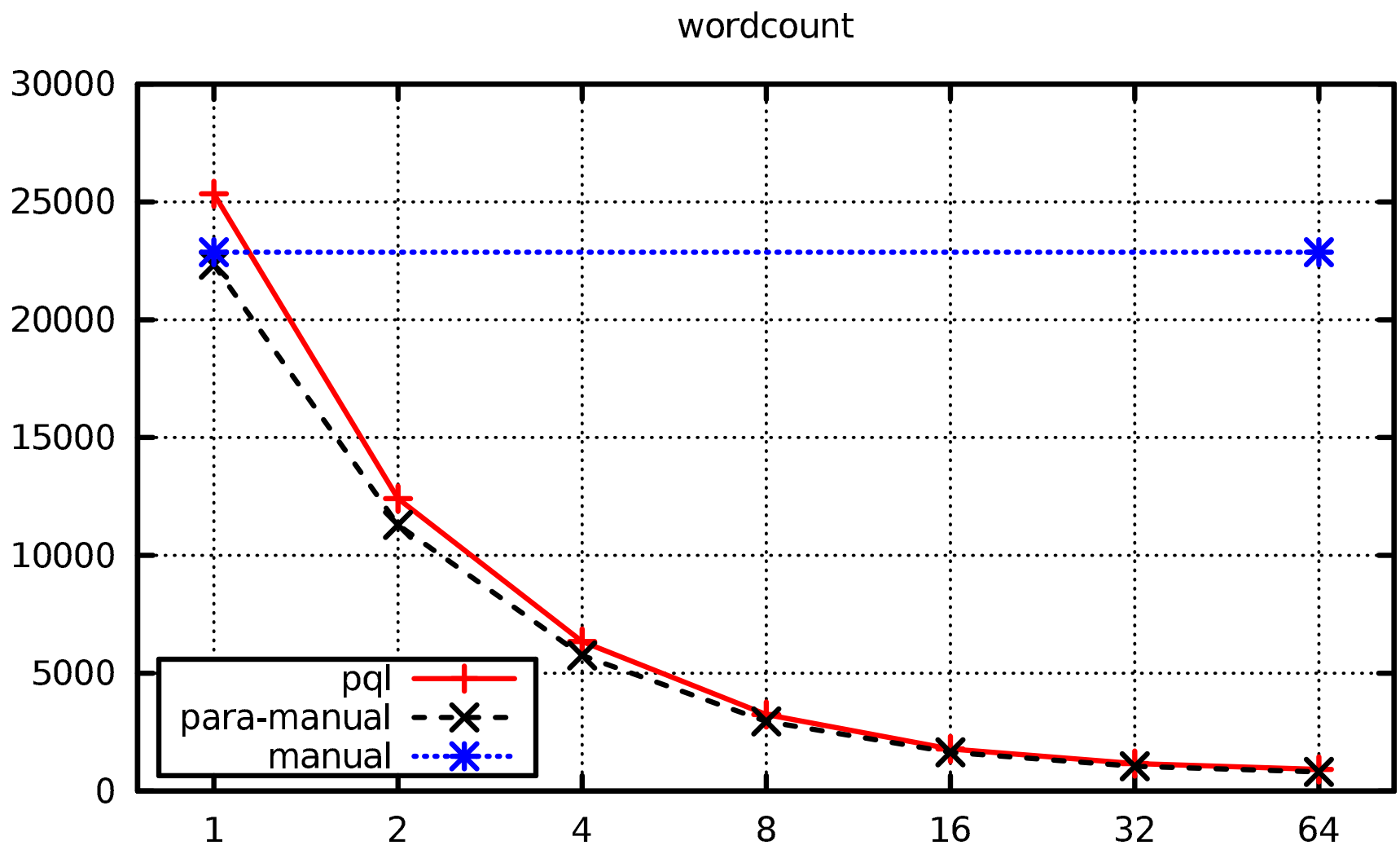
PERFORMANCE

- Benchmarks:
 - **bonus**: Salary computation
 - **threegrep**: String pattern search
 - **wordcount**: Word frequency aggregation in documents
 - **webgraph**: One-hop self-references in web graphs
- Hardware:
 - Intel Xeon 6×2 threads, 2.67 GHz, 24 GB RAM
 - Sun UltraSPARC 16×4 threads, 1.17 GHz, 32 GB RAM
- Methodology:
 - For each configuration: 3 warmup runs, 10 eval runs
- Running stock Sun JVM 1.6 with 2 GB Heap

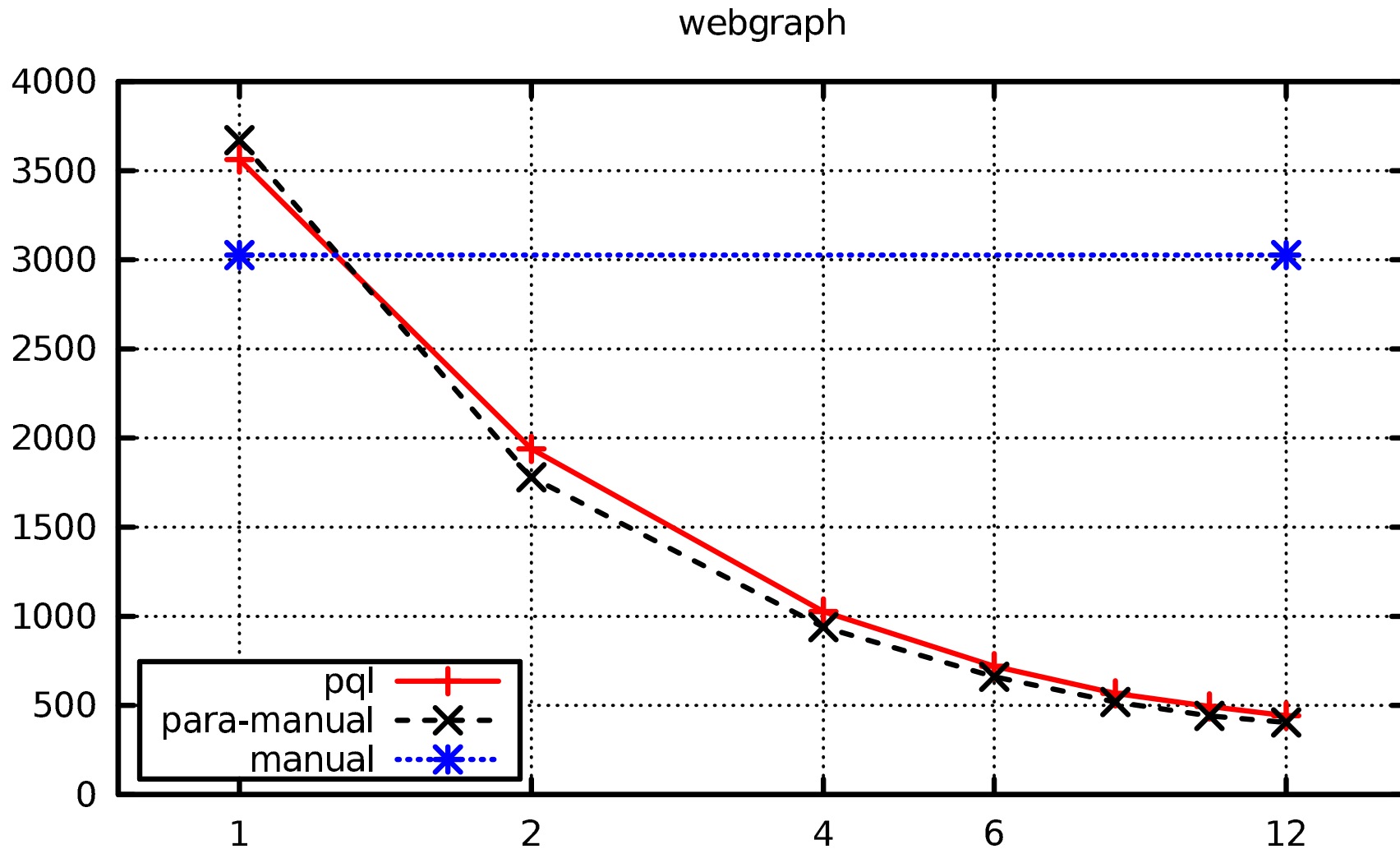
PERFORMANCE RESULTS: WORDCOUNT ON INTEL XEON



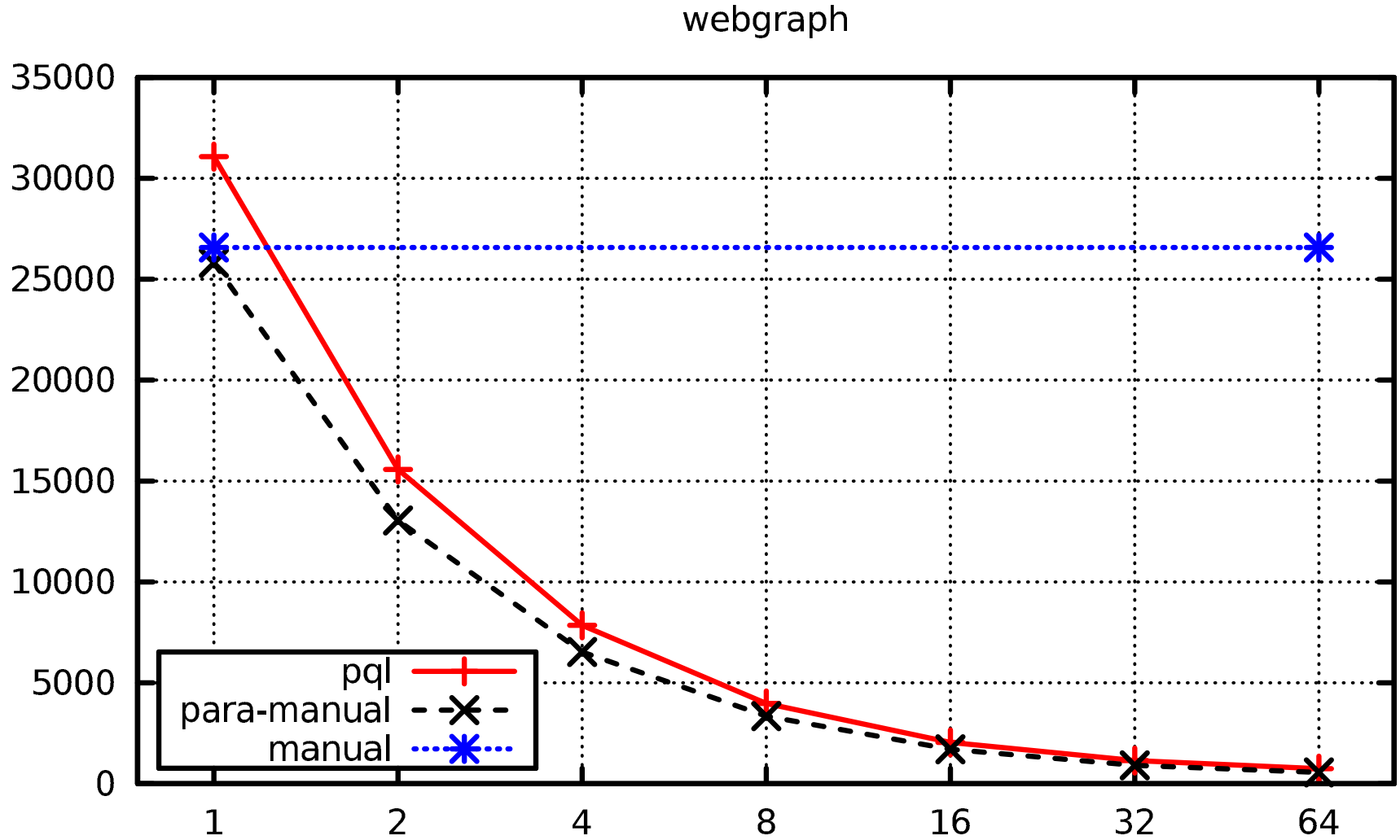
PERFORMANCE RESULTS: WORDCOUNT ON ULTRASPARC



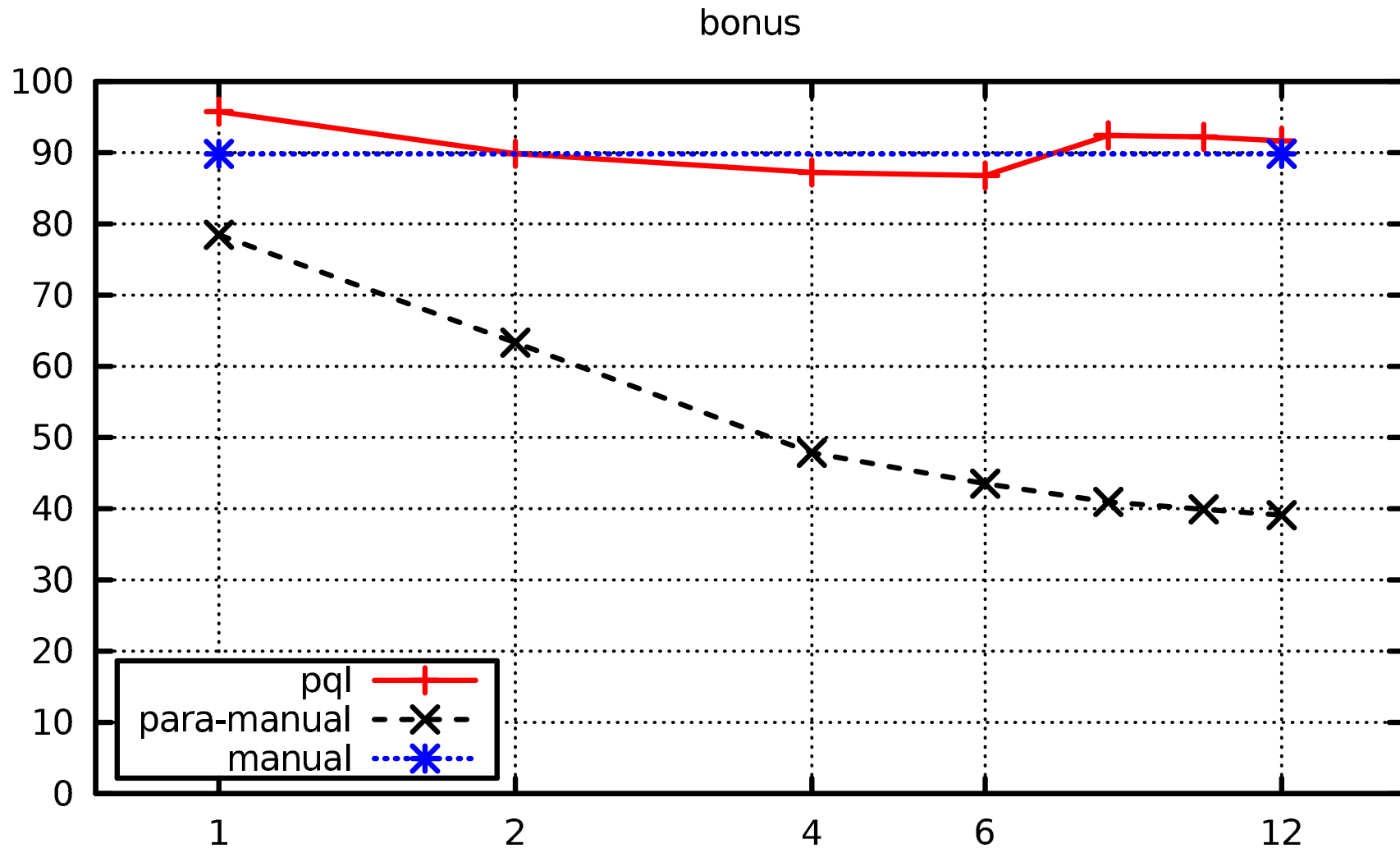
PERFORMANCE RESULTS: WEBGRAPH ON INTEL XEON



PERFORMANCE RESULTS: WEBGRAPH ON ULTRASPARC



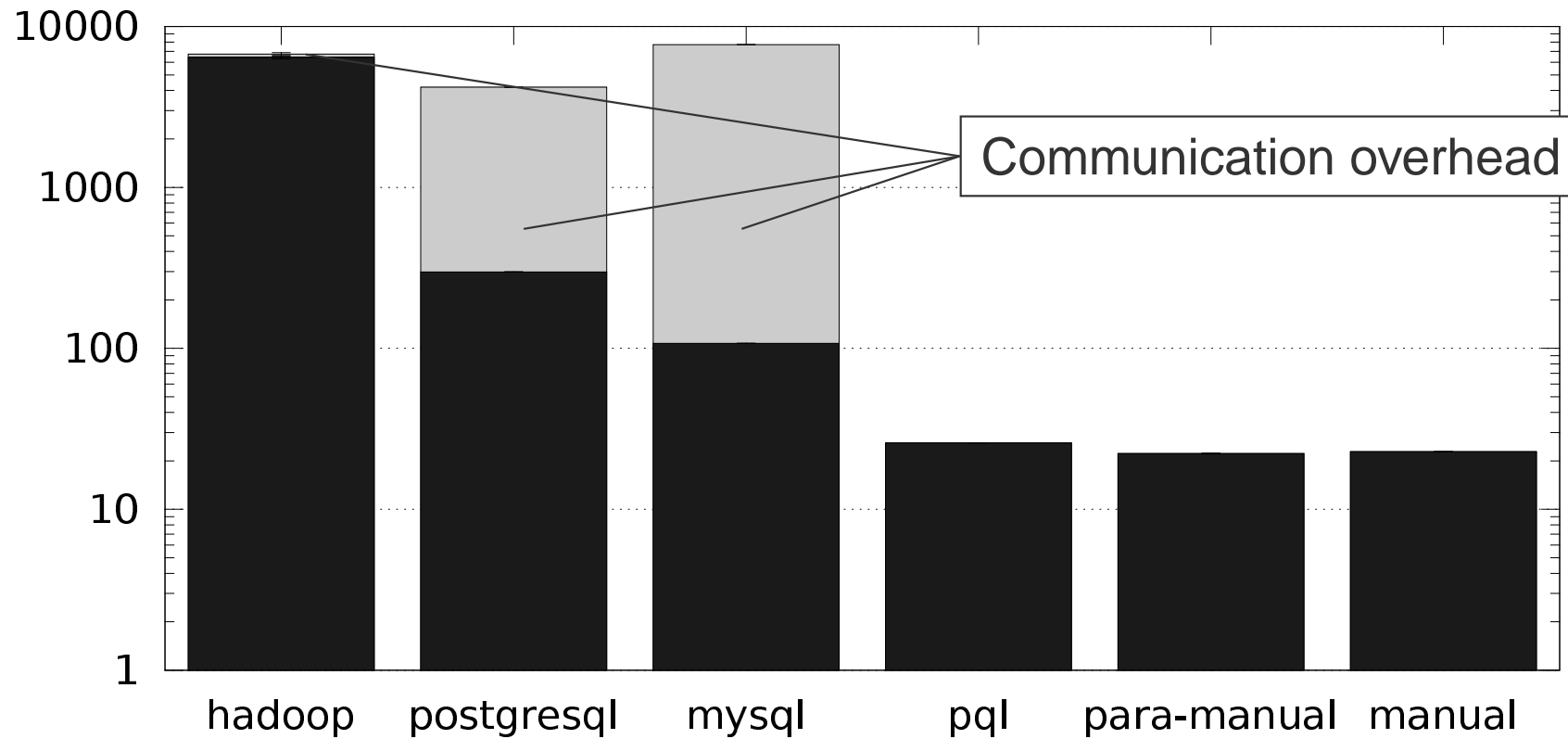
PERFORMANCE RESULTS: BONUS ON INTEL



EXISTING APPROACHES FOR JAVA

- SQL via JDBC: Similar queries, separate heap
- Hadoop: Java Map-Reduce framework

COMPARISON TO SQL AND HADOOP



(At $\frac{1}{10}$ of the usual benchmark size)

CONCISENESS

Total lines of code (including Java boilerplate):

benchmark	manual	manual- parallel	Hadoop	SQL	PQL
bonus	9	50	130	48	8
threegrep	9	46	60	21	6
webgraph	13	50	105	39	4
wordcount	8	98	93	38	4

PQL implementations are concise

WHAT THIS TALK DIDN'T COVER

- Intermediate language design
- Type inference
- Domain inference:
forall $x: a[x] == b[x]$: which x to check?
- More optimisations

Check the ECOOP'12 paper for details!

SUMMARY

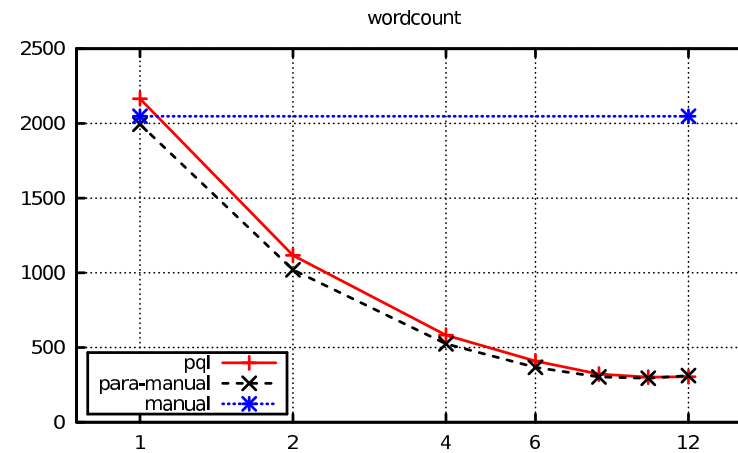
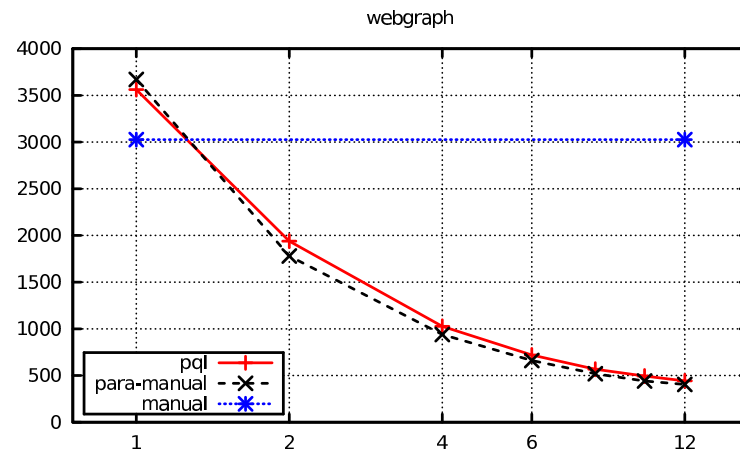
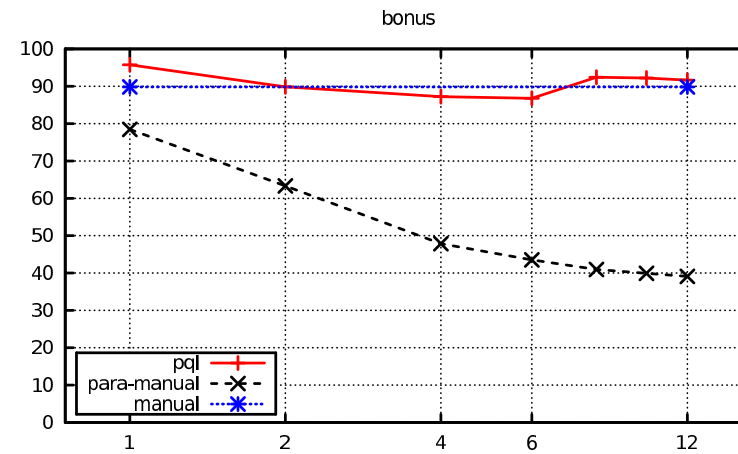
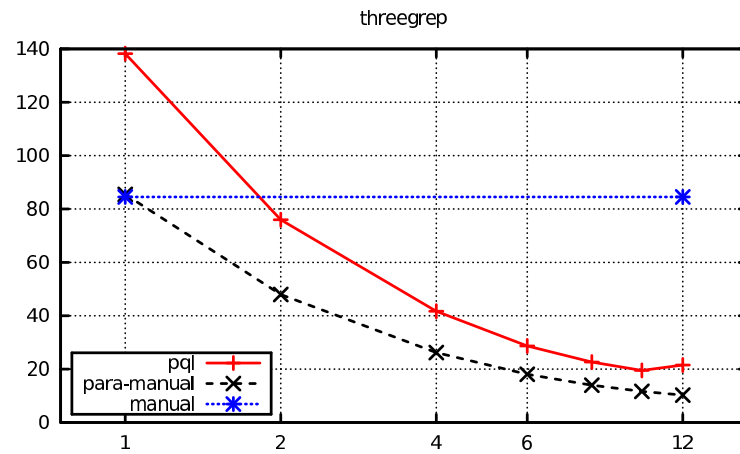
PQL/Java adds casual parallelism to Java through:

- Declarative query semantics
- Automatic parallelisation
- Strong parallel performance

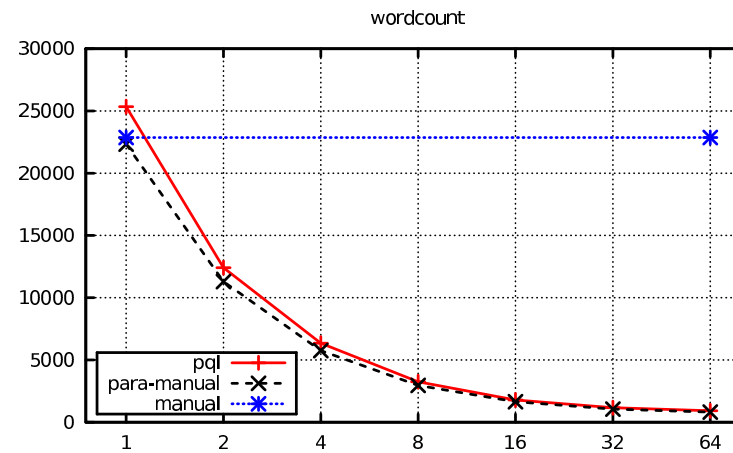
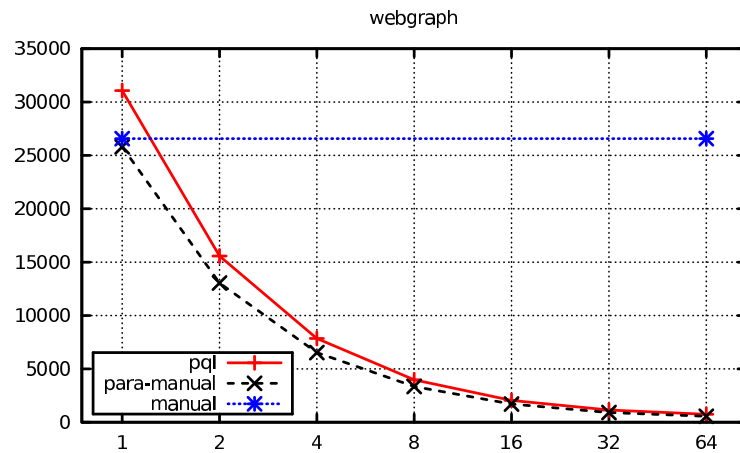
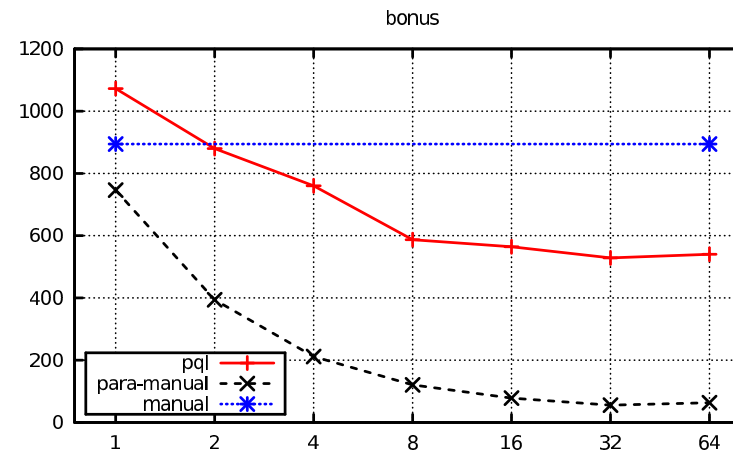
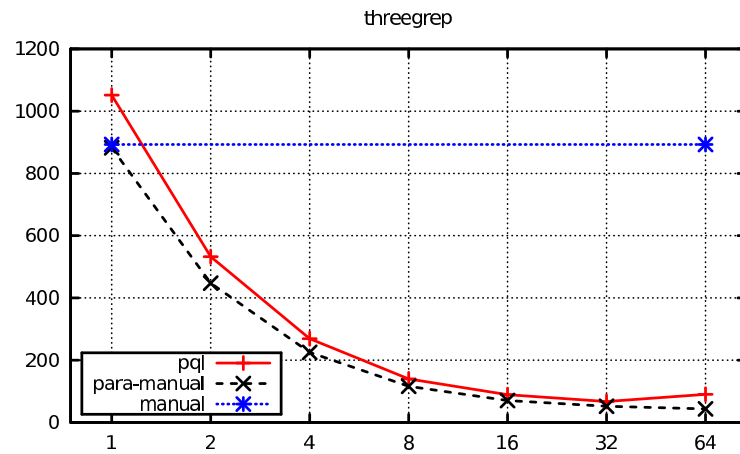
Available at <http://creichen.net/pql> (soon!)

Backup Slides

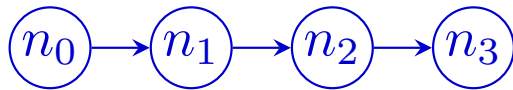
PERFORMANCE RESULTS: INTEL



PERFORMANCE RESULTS: SPARC



JAVA INTEGRATION: GRAPH REACHABILITY



```
new_edges = query(Map.find(from_node) == to_node):  
  !all_edges[from_node].contains(to_node) // new edge  
  && exists Node inter_node:  
    all_edges[from_node].contains(inter_node)  
    && new_edges[inter_node].contains(to_node);
```

JAVA INTEGRATION: GRAPH REACHABILITY

```
public Map<Node, Set<Node>> transitiveClosure(Map edges) {  
    Map<Node, Set<Node>> all_edges = edges.clone();  
    Map<Node, Set<Node>> new_edges = edges;  
    while (!new_edges.empty()) {  
        new_edges = query(Map.find(from_node) == to_node):  
            !all_edges[from_node].contains(to_node) // new edge  
            && exists Node inter_node:  
                all_edges[from_node].contains(inter_node)  
                && new_edges[inter_node].contains(to_node);  
        all_edges.putAll(new_edges);  
    }  
    return all_edges; }  
}
```