A New Java Runtime for a Parallel World

Christoph Reichenbach
University of Massachusetts
creichen@cs.umass.edu

Yannis Smaragdakis
University of Massachusetts
yannis@cs.umass.edu

Abstract

Parallelism is here to stay. Unfortunately, today's mainstream programming languages (such as Java) are not designed for easy parallelisation. We thus propose to extend Java with primitives for parallel queries, using a radically redesigned Java runtime system.

1. The challenge

Single CPUs with more than 50 cores are available today; at the current rate of growth, we can expect single applications to have access to thousands of cores within ten years. Exploiting such massive parallelism is challenging with today's mainstream languages (such as Java), since their imperative design and dynamic features make it hard to parallelise automatically. At the same time, manual parallelisation through thread library support is cumbersome and requires careful program design. We thus propose to extend Java with language features that are (a) easy to parallelise, (b) powerful, and (c) easy to use. We argue that meeting these challenges will require substantial changes to the Java runtime model.

First, consider some computations that are easy to parallelise and that any such language extensions should therefore support. The most promising candidates are computations that are 'embarrassingly parallel' in the sense of parallel computation. For these problems, increasing the number of CPUs by a factor of k will asymptotically speed up execution by that same factor k:

- *Combinatorial experiments*, wherein we try all options
- *SQL-like queries, selection and filtering*
- *Term substitution* of free variables in algebraic terms
- *Searching in an unsorted array*

These computations will likely only make up part of any interesting program, but we can speed them up arbitrarily (for a suitably large input size): whenever we double the number of cores in our system, we halve the time needed for these queries — as long as our language and runtime provide support for them.

All of the above computations, including SQL selections with filters and joins, fit into the model of first-order queries. Such queries have the form 'give me all $\langle x_1, \ldots x_n \rangle$ that make property $p(x_1, \ldots, x_n)$ hold, where p is a first-order logical formula. Thus, these queries would be a good first approximation for the language features we are looking for. The real challenge then lies in providing runtime support for such queries.

2. Overview

Parallelising queries such as the above is impractical with the standard Java heap model. To see this, let us try to implement a fast parallel implementation of the 'contains' check for Java containers, as in Figure 1.

If datalist is an ArrayList, it is easy to see how we can parallelise this task:

```
class Data { int value; String name; ... }
Data key = new Data(...);

List<Data> datalist = ...;
return data.contains(key);
```

Figure 1. Parallelisable example program

- *Fan out:* We assign each core a portion of the array that is roughly the same in size for all cores, avoiding overlap.
- *Execute:* Our cores now compute local results, which may be **true** or **false**.
- *Fan in:* Finally, the cores write back their result. In our case, the result is a boolean flag, so each core would only write **true** if it had found the result and otherwise leave the result untouched (defaulting to **false**).

However, this parallelisation is critically dependent on the structure of our list. If datalist is not an ArrayList but a LinkedList, it fails: in a linked list we only see the first (and perhaps the last) element; to find intermediate elements to fan out on, we have to chase pointers across the heap. This problem generalises to other queries we might be interested in, such as our term substitution example: if we start with a root term object, we must again chase pointers to find all nodes that make up the term. This is something we cannot parallelise — unless we change the Java memory model.

3. Heap design for quick fan-out

What if we represent objects not as memory blocks with pointers, but instead as records in relational tables, indexed by unique keys?

In that case, we can fan out on our list (or on any other data structure) easily, as long as all the list nodes are stored in the same table. We can then exploit the straightforward table layout exactly as we did for the parallel ArrayList.contains check.

For representing LinkedLists, the array would be an array of LinkedList.Entry, where LinkedList.Entry is the inner class that represents list nodes (i.e., each LinkedList.Entry contains a value, a predecessor reference, and a successor reference.) As a first approximation, this array of LinkedList.Entry objects might contain all instances of LinkedList.Entry on the heap.

This idea gives us a notion of 'classes-as-tables', and it works well if we want to query all instances of LinkedList.Entry. While querying all instances of a class may be useful, we need more finesse for the LinkedList example, for two reasons: first, if we have multiple linked lists, fanning out over all LinkedList.Entry nodes would also fan out over nodes in other lists, which would slow us down; second, a linked list may contain entries of many types — a linked list of Integer will contain LinkedList.Entry<Integer> nodes, and a linked list of strings will contain LinkedList.Entry<String> nodes, and we must not confuse these two.

Fortunately, the existing design of the Java language and libraries offers a convenient solution: in existing implementations (such as the GNU Classpath implementation of `java.util.LinkedList`), inner helper nodes such as `LinkedList.Entry<Integer>` are typically inner classes of limited visibility (**private**, in the case of Classpath). This means that such helper nodes are only ever instantiated within their surrounding class, either statically or in the presence of an 'owning' object. We can thus safely give either the static class or the owning object a table for each such inner class, and fan out over exactly this table.

3.1 Accommodating sequential code

The price we pay for translating all objects into rows in a relational table representation is that field lookup is no longer straightforward. In the standard Java heap model, we look up a field by interpreting an object reference `obj` as a pointer p, a field `f` as a constant offset δ to that pointer, and a field reference `obj.f` to a read from (or write to) memory address $p+\delta$. In a relational representation, accessing a field requires an extra indirection. Consider class `Data` from Figure 1, with fields **int** `value` and `String name`. We might represent elements of this class in a table such as

ID (key)	value	name
`0x1fba`	0	`"foo"`
`0x0052`	23	`"bar"`
`0x0e08`	17	`"quux"`
...	...	...

If we now want to look up a field from this table, e.g. `key.name`, we have to look up the object ID (such as `0x0052`) in this table before we can access the correct field. We can speed up the lookup by keeping the table sorted, but that comes at extra cost when creating new entries; we can speed up the search by allocating extra cores for table lookup, but this speedup is bounded by the number of CPUs and thread execution overhead.

We thus propose two additional strategies for combatting the overhead of this representation:

- *On-demand representation translation*, wherein we initially represent objects with the default Java representation (as much as possible) and only translate them the first time we need them for a query, and
- *Field caches*, which remember 'popular' fields (by a suitable heuristic) in a form that is easier to access.

With field caches, we represent each Java object as a pointer to a record, which in turn contains the usual Java object header (virtual function tables etc.) together with a fixed-size field cache array. This array stores recently modified fields and any information needed by the access heuristic. Meanwhile, the object pointer simultaneously serves as the ID (or primary key) to the object table.

To gauge the overhead of this strategy for strictly sequential code, we implemented a small number of C micro-benchmarks and ran them with two memory models. The first memory model, *simple*, represents each object as an array and accesses fields with fixed offset into that array. We expect this memory model to behave equivalently to record field access. The second model, *cached*, represents each object in two ways: first in a hash table laid out similarly to our `Data` table above, and secondly in a field cache for up to three fields. Each field cache read hashes the field offset to a cache location, checks whether the field is currently in the cache, and falls back to the hash table (updating the field cache) as needed. For updating fields, we use a write-through strategy.

We applied both memory models to four problems: *qsort* (quicksort), *1d-puzzle* (solving a one-dimensional puzzle by brute force), *2d-puzzle* (as *1d-puzzle* but in two dimensions), and *read*, which allocates a large number of objects (using more than two orders of magnitude more memory than offered by the L1 cache) and measured read performance. Of our benchmarks, *quicksort* and *1d-puzzle* used two fields (causing no field cache conflicts), while *2d-puzzle* used four fields and *read* used 20 fields, causing frequent field cache conflicts.

We compiled these programs with gcc 4.4 and ran them on an Intel Core 2 6600 clocked at 2.40GHz. None of our benchmarks used any form of parallelism. The table below summarises the end-to-end run-time of 100 runs of each benchmark:

setup	qsort	1d-puzzle	2d-puzzle	read
simple	8.547s	94.664s	14.074s	2.959s
cached	15.595	107.559	15.416	4.489s
slowdown	1.83	1.14	1.10	1.52

The slowdown we measured was thus between 10% and 83%. Note that both of the puzzles (in either heap model) fit comfortably into the L1 cache for both representations, despite field cache conflicts. **quicksort** and **read** did not, which may explain their less satisfying performance.

Note that this experiment did not take advantage of any parallel optimisations, such as field prefetching, and deliberately assumed that we were representing the entire heap in relational form. Thus, there may still be considerable room for improvement left in the sequential case. Of course, in practice a slowdown of 83% of half of a program may still be acceptable if, in return, we speed up the other half by a factor that is only bounded by the number of cores we can fit into our system.

3.2 Other challenges

Supporting our heap design requires further design considerations, some of which we list below.

Memory management. Once we complete a parallel computation, we have to 'fan in' the results. Boolean result are straightforward, as we showed. However, when we fan in structured results — such as a term after variable substitution — we need to make sure that we represent result in the same way that we represent other heap objects. This is nontrivial, as all cores may be allocating objects in parallel, especially since object allocation in our model this means allocating both table rows and field caches for each object. Clearly, we must avoid any form of locking. To facilitate such parallel allocation, we therefore suggest the use of core-local field cache object allocators together with partitioned hash tables, where each core is assigned a partition in the result table such that the core-local field cache object allocator only generates objects at addresses that map into the core's partition: this ensures that there will be no overlap on tables or on the 'regular' heap.

Subclasses. We can implement subclasses by sharing common fields in the superclass table and storing only 'new' fields in subclass tables. However, this may yield less than perfect locality. An alternative design would be to heuristically pad each table row in the superclass table, and use this padding to store the fields of any subclasses (whenever possible).

4. Conclusion

To provide facilities for fast, expressive, and parallelisable queries in Java, we propose a redesigned Java run-time system, built expressly for the needs of parallelisation. While such a transition comes at a cost to sequential code, we argue that this cost can be offset by a number of techniques, such as field caches and on-demand representation translation. We expect that we can thus, at a small runtime cost, provide both significant increases in expressive power and scalable parallelisation to Java programmers.