Working Paper Series
ISSN 1177-777X

THE JSTAR LANGUAGE PHILOSOPHY

Mark Utting, Min-Hsien Weng, and John G. Cleary

Working Paper: 06/2013
November 11, 2013

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

The JStar Language Philosophy

Mark Utting[a,], Min-Hsien Weng[a], John G. Cleary[a]

[a] *Department of Computer Science, FCMS, The University of Waikato, Hamilton, New Zealand*

Abstract

This paper introduces the JStar parallel programming language, which is a Java-based declarative language aimed at discouraging sequential programming, encouraging massively parallel programming, and giving the compiler and runtime maximum freedom to try alternative parallelisation strategies. We describe the execution semantics and runtime support of the language, several optimisations and parallelism strategies, with some benchmark results.

Keywords: Parallel programming models, architecture independence, JStar, Java, Datalog, Linda-like languages.

1. The Goals of JStar

JStar is a new declarative programming language designed for implicit parallel programming [6]. The language semantics is Datalog with negation, plus an explicit *causality ordering* that defines a *local stratification ordering*, which both ensures that programs have a well-defined semantics and loosely constrains the execution order [6].

Broadly, the main aim of JStar is to discourage sequential programming, and instead encourage a programming style that has massive amounts of implicit parallelism, so that by choosing appropriate options, a compiler can generate several different kinds of implementations, including sequential, multi-core, GPU, etc. More precisely, we had four main design goals for JStar, which are described in the following subsections.

1.1. Raise the Abstraction Level

The time taken to write a program is roughly proportional to the size of the program in lines of code [10]. So to increase programmer productivity, we would like to be able to express a given program in a more concise form. To our minds, most current programming languages are already too verbose, requiring many control flow and data representation issues to be over-specified. And making a program parallel usually requires adding even more code. Our goal for JStar

Email addresses: `marku@waikato.ac.nz` (Mark Utting), `mw169@waikato.ac.nz` (Min-Hsien Weng), `jcleary@waikato.ac.nz` (John G. Cleary)

is that there should be no code overhead for making a program parallel, and if possible, JStar programs should be more concise than say, an equivalent Java program.

To achieve this, JStar uses the expressions of the Eclipse XText framework (essentially Java expressions with type inference and lambda expressions), plus a concise one-line notation for defining relational *tables*, and a simple *foreach* notation for defining rules.

1.2. Avoid Mutable Data

Most of the problems in parallel programming arise from multiple theads updating shared variables. In JStar we ban mutable variables,[1] and take a more declarative approach where each computation rule takes one or more tuples as input and produces new tuples as output. This is somewhat similar to functional programming, but avoids the 'plumbing problems' inherent in that style[2] by using a single global database, like the assert database of Prolog. Computation rules can query this database and can add their output tuples to it, but they cannot mutate or delete tuples. Of course, the language semantics allows garbage collection of tuples that will never be used again.

1.3. Make Parallelism the Default

Modern computers offer an increasing number of cores, so all modern programs should offer scalable parallelism, to take advantage of the available hardware. In JStar, we want parallel execution to be the default, so that a programmer has to take extra measures if they wish to constrain the execution to be sequential. We aim to discourage sequential programming (no **while** loops or sequential **for** loops), in order to encourage the programmer to think in terms of parallel computation. JStar does allow **for** loops to be written within a rule, but since there are no mutable variables, every iteration of the loop body is independent, which allows greater parallelism.

To replace some common uses of sequential loops, JStar supports reduce and scan operations with user-defined operators. Our goal is that JStar programs specify the minimal constraints on the execution order that ensure that the calculations are correct. Furthermore, part of this design goal is that all programs should have *deterministic parallel semantics* [3], meaning that the output of the program is independent of the parallelism strategy that is used.

1.4. Late commitment to data structures

In imperative parallel programs, the choice of data structures is often strongly linked with the kind of parallelism that the program uses. This means that making major changes to the parallelism structure of the program often requires major changes to the data structures too, which makes it hard to experiment

[1] They may appear only in 'unsafe' code blocks, which are used to implement system rules.

[2] That is, one frequently has to feed an input or output parameter down through many levels of functions in order to get access to it at the point required.

with many alternative parallelism approaches. To avoid this dependency, we want programs to be written using neutral high-level data structures (relations), which can be transformed into efficient implementation-oriented data structures *after* the parallelism structure of the program has been designed, and after we know how the program queries each relational table.

This relational approach gives the language much more implementation flexibility than just abstract data types or interfaces, since we can perform static analysis on the queries that are performed (part of the query term is typically written using a boolean lambda expression) before deciding how to represent the data, which fields should be indexed, what data structures to use for each index, etc. Currently we just generate default indexes and data structures for each relation, then allow the programmer to override those choices via runtime flags.

1.5. Current Status

The current implementation of JStar (v2.0) is based on the Eclipse XText domain-specific language environment.[3] This makes it easy to create a modern Eclipse-based IDE for editing the language, plus a compiler that generates Java code. We have added compiler flags for generating either sequential code, or parallel code using the Java Fork/Join framework [9]. JStar also supports:

- a simple graph visualizer for viewing aspects of the partial order over tuples that controls the parallelism;
- a connection to several alternative *Satisfiability Modulo Theories* (SMT) theorem provers[4] for proving that rules are consistent with the causality orderings declared by the programmer, and that tuple invariants are preserved;
- a logging system for recording usage statistics about each table during a program run, and tools to visualise those logs as annotated dependency graphs of the program execution. This is a useful basis for choosing parallelisation strategies.

This paper reports on our current work on automatic parallelisation of JStar programs for multicore CPUs, but we have also explored implementations of a few example Starlog programs on cluster computers [7], on GPUs [2], and have achieved good performance and scalability.

2. Programmer Workflow

The left side of Figure 1 shows a simplified workflow for developing an imperative parallel program [16]. The source code of the program is conceptually

[3] See `http://www.eclipse.org/Xtext`.
[4] See `http://www.smtlib.org` for example SMT tools.

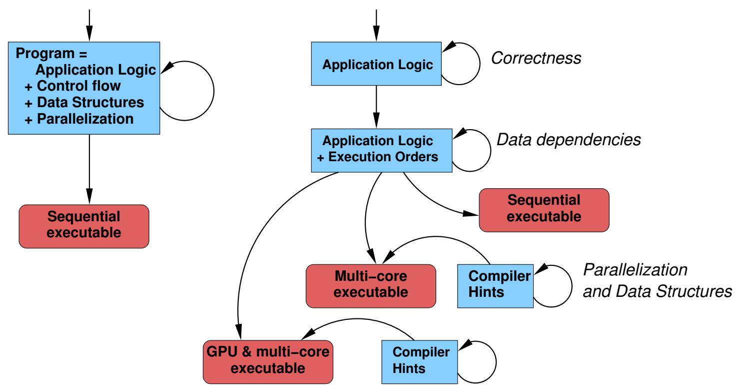

Figure 1: Parallel program development workflow: traditional (left) versus JStar (right).

one large integrated piece of text (albeit spread over several files) that defines the computation of the program, its flow of control (parallel and serial), its data structures, locking strategies, etc. These aspects are all intertwined in the source code, and the programmer must get them all correct before the program can be executed.

In contrast, in the JStar workflow (right hand side of Figure 1) these concerns of the programmer are separated out into four different stages:

1. **Application Logic.** At this stage the programmer defines just the schemas for all the tuples that will be computed, and the rules for creating those tuples. This defines the input-output functionality of the program, but leaves parallelism, control flow, and data structures, largely unspecified. The programmer's main concern is the functionality and correctness of the program. The program is executable, though not very efficient, so the programmer can test the functionality of the program and check that it is correct, before starting to work on the parallelism and efficiency aspects. The main tools needed at this stage are an IDE with good support for editing and refactoring the program, plus good support tools for unit testing and system testing.

2. **Possible Execution Orderings.** At this stage the programmer thinks about the dependencies between the rules and tuples, and defines the weakest possible orderings between them, in order to allow for the largest range of parallel execution strategies. These causality ordering declarations are part of the program source code, since they are usually architecture-independent. Static analysis and automated theorem proving are used to check that the dependency orderings proposed by the programmer are consistent with the computation rules in the program. Visualization tools are also useful at this stage, to show the dependencies as directed graphs.

3. **Parallelism Strategy.** For each target architecture, the programmer now

designs a set of instructions to the compiler saying which rules should be run in parallel, whether each set of tuples should be partitioned, duplicated or shared across the different cores or computers (for distributed implementations), and how the communication should be implemented. These instructions are separate from the program (with a different set of instructions for each target architecture), and are used by the compiler to transform the original declarative JStar program into an architecture-specific parallel or distributed program. Since the set of output tuples of a JStar program is independent of the choice of parallelism, this stage can change the efficiency of the program but cannot change its correctness (input-output behaviour is preserved, except that output tuples may be produced in a different order). The main tools needed at this stage are parallel profiling tools to report on the performance of tasks, statistics about communication between tasks etc.

4. **Data Structures.** Once it is known how each task will access the tuples, appropriate data structures, indexes and buffering strategies can be chosen. These choices are also stored separately from the program source, and are used as hints to the compiler to say what code should be generated for each kind of tuple. The main tools needed are profiling and recording tools to analyze the performance of each data structure plus metrics about how it is used [5]. This stage is similar to performance tuning of relational databases, except that it is tuning an in-memory database.

Because the architecture-dependent compiler hints are separate from the program source code, it is easy to experiment with alternative implementations – one can simply design multiple sets of compiler-directive files, one for an efficient sequential implementation, several different parallelization strategies for a multi-core implementation, and then run the compiler with each of those compiler-hint files and benchmark the resulting programs to see which approach is more efficient on resources. This encourages an empirical approach to the parallelization of programs. It is even possible that different kinds of programmers may perform different stages, such as an application domain expert performing stage 1, while a parallel-performance expert performs stages 2-4.

3. The JStar Language

This section gives an informal overview of the JStar v2 language and its execution semantics. The formal semantics, and a proof of the correctness of all the possible parallel evaluation strategies, can be found elsewhere [6]. In brief, it is equivalent to Datalog with negation, functors, and a local-stratification ordering (which we derive from a programmer-supplied causality ordering over tuples).

JStar uses a very different programming paradigm to most existing languages. It is a *relational* programming language. The key idea is that all data manipulated by a program is stored in in-memory relations/tables, and rules can add tuples to these tables but cannot update or delete existing tuples.

frame	x	y	dx	dy
0	10	10	150	0
1	160	10	150	0
2	310	10	150	0
3	460	10	0	10
4	460	20	0	10
5	460	30	-150	0
6	310	30	-150	0
7	160	30	-150	0

Ship

Figure 2: Ship table, showing a ship moving right, then down, then left.

If we want to record data that changes over time, such as the position of a ship in a Space Invaders game, then we must add timestamp information to each tuple in the database. For example, Figure 2 shows a `Ship` table that records the movements of a single ship over 8 frames It first goes across the screen to the right in 150 pixel jumps, then descends slowly several times, then moves to the left in 150 pixel jumps.

Each tuple in a table is typically implemented as an immutable Java object with a fixed set of named fields, corresponding to the columns of the table. This Ship table could be declared by the following command, which creates a Java class called Ship, plus another class called ShipTable with various query and lookup methods. The `->` is a shorthand that indicates that the `frame` field is a primary key, so the ShipTable class has an invariant that only one Ship can exist within each frame value. The **orderby** list will be discussed in Section 4.

```
table Ship(int frame -> int x, int y, int dx, int dy) orderby (Int, seq frame)
```

When a tuple is queried or created, the field values can be specified by position, or by name (using the `[...]` lambda expressions of Xbase). Here are several equivalent expressions that create a tuple equal to the first tuple in Figure 2. We also generate a *builder* class for each table, so that a `copy` method can take an existing (immutable) tuple, update a few fields and create a new tuple.

```
new Ship(0,10,10,150,0) // by position (in order)
new Ship() [frame=0; x=10; dx=150; y=10; dy=0]  // by name
new Ship() [x=10; dx=150; y=10] // use default values for frame and dy.
```

In addition to these tables, the other main part of a JStar program is a set of *rules* that add new tuples to the tables. Each rule inspects the existing database, makes calculations and decisions, and can then add tuples to one or more tables. Here is a simplistic rule that always moves the Ship to the right by 150 pixels.

```
foreach (Ship s) { put new Ship(s.frame+1, s.x+150, s.y, s.dx, s.dy) }
```

JStar uses an improved incremental version of the *pseudo-naive* execution algorithm [12, 6], so when new tuples are added to the database, they are placed into a temporary area called the *Delta Set*. Each execution step removes one or more tuples from the Delta Set, adds them into the appropriate tables in the main database (Gamma), and then executes all the rules that have those tuples as inputs. This is a *bottom-up* [15] or *forward-chaining* execution mechanism, similar to that used by some expert system and planning engines.

Event-driven programming with external input tuples fits elegantly into this framework – the input tuples are added to the Delta Set, and can then trigger various rules before being stored into a table (or discarded if they are no longer needed). Similarly, some tuples generated by the program can be requests for external actions, such as reading or updating files – such actions are performed when those tuples are taken out of the Delta Set.

The above rule is triggered unconditionally by every Ship tuple, so it will be executed for every Ship tuple that is added to the database, which effectively creates an infinite loop that keeps moving the Ship infinitely far to the right! This is perhaps not quite what we want, so let's change the rule so that it moves the ship to the right only when its `x` position is less than 400 pixels:

```
foreach (Ship s) {
  if (s.x < 400) { put new Ship(s.frame+1, s.x+150, s.y, s.dx, s.dy) }
}
```

This bottom-up execution mechanism has the potential for lots of parallelism. At first glance, it looks like every tuple in the Delta Set could be executed in parallel. However, this is not always safe for rules that contain negative or aggregate queries. The next section discusses constraints on the rules and on the parallel execution algorithm to ensure that programs have sensible and deterministic semantics.

4. The Law of Causality

In JStar, timestamps are used to record the passage of time, so we usually include some explicit timestamp fields in each tuple – for the Ship table the `frame` field is the timestamp. In some other tables, the timestamp may be comprised of several fields. For example, if we want to print a 2D table of numbers, we might define a timestamp based on the *line number* then the *column number* so that the outputs are printed in the desired order.

Just like in the real universe, it is a fundamental law of JStar that rules can affect the future, but they are not allowed to change the past! So the tuples that are 'put' into the database by a rule must have later timestamps than all the input tuples that are read by the rule. This is the law of *causality*. The example rules we saw for moving a Ship to the right all satisfy the law of causality, because the new Ship tuple is added to a later frame than the input Ship tuple.

The causality law is important because JStar allows *negative* and *aggregate* queries of the database as well as positive queries. A negative query checks that a given set of tuples are *not* in the database, while aggregate queries can count or sum or combine tuples in various ways. Without the causality restriction, the execution of future rules could affect the results of such queries and invalidate calculations that depend on the query results.

A more precise definition of the causality law is that a rule that puts a tuple with timestamp T into the database can only perform positive queries with timestamps $\leq T$, and negative or aggregate queries with timestamps $< T$. This is the same as the *local stratification* requirement for the sound execution of Datalog programs [11, 6]. We use SMT solvers (automatic theorem provers similar to SAT solvers) to check that each rule is consistent with the programmer-supplied causality ordering. For example, given a rule like:

```
foreach (Trigger trig) {
  if (Cond) {
    val tuple1 = new Tuple1(args1)
    put tuple1
  } else {
    val q1 = get min Tuple1(queryArgs)
    val tuple2 = new Tuple2(args2)
    put tuple2
  }
}
```

we send one causality proof obligation to the SMT solver for each **put** command to ensure that the new tuple is being added into the future (or present) part of the database (we use `orderby(`T`)` to mean that the tuple T is unfolded into its **orderby** list, so that only those named fields are used by the causality ordering):

```
1. inv(trig) and Cond and inv(tuple1)
   ⟹ orderby(trig) ≤ orderby(tuple1)
2. inv(trig) and ¬Cond and inv(q1) and inv(tuple2)
   ⟹ orderby(trig) ≤ orderby(tuple2)
```

and one to ensure that the query timestamp is strictly before the trigger tuple (which means that the result of the query is now fixed):

```
3. inv(trig) and not(Cond)
   ⟹ orderby(Tuple1(queryArgs)) < orderby(trig)
```

If the SMT solver cannot prove one of these theorems, the relevant statement is marked with a warning message, and the programmer is strongly recommended to change the program (eg. strengthen invariants, make queries more specific, or change the **orderby** clauses) so that the solver can prove that the ordering relationship is satisfied.

5. Parallelisation Strategies in the JStar Compiler

The JStar compiler translates each JStar program into standard Java source code, which can then be compiled with a Java compiler and executed. The compiler generates parallel Java code and data structures by default, or can generate sequential code and data structures if the `-sequential` compiler flag is supplied. This section briefly describes the parallelisation strategies that the

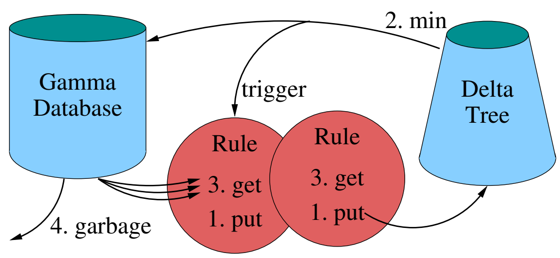

Figure 3: The lifecycle of a typical JStar tuple.

compiler uses, some important optimisations that it supports, and how users can override the default choice of data structure.

As mentioned above, the lifecycle of a typical tuple in a JStar program is as follows (see Fig. 3):

1. A rule (or an initial **put** command) creates the tuple, which is then inserted into the *Delta* set to await its turn for processing.
2. The tuple is taken out of the Delta set (in an order that respects the causality ordering of the program), is used to trigger any applicable rules, and is put into the *Gamma* database that (conceptually) stores all tuples generated by the program.
3. When other rules execute, they may query tables in the Gamma database, and this tuple may be returned as the result of a query.
4. If program analysis makes it possible to determine that this tuple can never participate in future queries, then it can be removed from the Gamma database and garbage collected. Currently, this program analysis is not automated, so we simply retain all tuples, or use manual lifetime hints from the user to determine when tuples can be discarded.

Our current implementation uses a very simple parallelisation strategy built on top of the Java 7 Fork/Join framework. It treats the Delta set as an event queue, ordered by the causality ordering. At each execution step, it takes all *minimal* tuples out of the Delta set, and executes all those tuples in parallel. More precisely, the Delta set is organised as a single tree, containing tuples from many tables, sorted lexicographically by the **orderby** lists of those tables. That is, the i^{th} level of the Delta tree is sorted according to the i^{th} entries of the **orderby** lists. If the i^{th} entry of an **orderby** list is `seq` e then the subtrees of the corresponding node of the Delta tree are sorted sequentially by the value of field e; if it is `par` e then the subtrees are unordered so can be executed in parallel; and if it is a capitalised literal name then it is sorted according to a partial order specified by explicit **order** declarations in the JStar program (eg. `order Req < PvWatts < SumMonth` in Fig. 4). The leaves of the Delta tree contain sets of tuples–the tuples within one of those sets are all in the same equivalence class with respect to the Delta ordering, so can be executed in parallel.

For example, if the Ship table were declared as:

```
table Ship(int frame, int x, int y, int dx, int dy) orderby (Int, seq frame);
```

then multiple Ships would be allowed within each frame, but all those Ships would be equivalent within the causality ordering, since the **orderby** list only uses the `frame` field and ignores the other fields. The root node of the Delta tree (level 0) would have a named subtree called 'Int', level 1 would be sorted sequentially by the `frame` field, and each level 2 node would be a leaf node that contains a set of all the Ship tuples within the given frame. If we had 11 Ship tuples within frame 18, then when execution reaches that frame, 11 fork/join tasks will be created, and each of those tasks will fire one or more rules, which will in turn insert new tuples into the future area of the Delta tree.

This implementation uses the Delta tree as a multi-level priority queue, so it is important to be able to quickly find the minimum node in the tree, and to remove duplicate tuples that are inserted into the tree. The nodes of the Delta tree that contain named branches are implemented as a linear array of subtrees, indexed by a total ordering of the **order** relationship at that level. The nodes of the Delta tree that contain sorted integer indexes typically use a Java `SortedMap` to order the subtrees,[5] with the default implementation being a Java `TreeMap<Integer,DeltaNode>` when generating sequential code, or a Java `ConcurrentSkipListMap<Integer, DeltaNode>` when generating parallel code.

The Gamma database contains a separate data structure for each table, and the default implementation of this uses some kind of `NavigableSet`, so that queries of any ordered subset of the tuples can be performed reasonably efficiently. When generating parallel code, `ConcurrentSkipListSet` is used, and when generating sequential code, `TreeSet` is used.

5.1. Program Optimisations

Some tuples are never used as the triggers of rules - they are used only by queries from within rules. In this case, it is not necessary to send the tuples through the Delta tree - they can be put directly into the Gamma database. Even when a tuple does trigger a rule, it can be executed immediately if that rule does not query the database or contain 'unsafe' code (which may have external side effects). The compiler supports this optimisation via a `-noDelta T` flag, which generates code to put each new T tuple directly into Gamma, and immediately fire any rules that have it as a trigger. As we shall see in the next section, this optimisation can dramatically speed up programs that generate a lot of non-trigger tuples.

A complementary optimisation is `-noGamma T`, which omits the insertion of tuples from table T into Gamma. This is useful when tuples are used only as triggers, and are never queried. This optimisation typically has only a minor effect on speed, but does help to reduce the active heap size.

[5] A priority-queue is not sufficient, because we also need to remove duplicate tuples as they are inserted.

5.2. Additional Parallelism

Note that this parallel implementation does not take advantage of all the potential parallelism in a JStar program. Even if a tuple triggers more than one rule, we create only one task for that tuple - we could create one task per rule that is triggered. Also, within a rule, any loop that does not use a reducer object is known to have independent loop bodies, so these could be executed in parallel. Loops that do involve a reducer object could also be executed in parallel, with a tree-based pass to combine the final reducer results. Finally, the semantics of JStar allows for many different parallel execution orders when extracting tuples out of the Delta set [6]. This paper does not investigate those additional parallelisation opportunites - it just reports on the effectiveness of the simple all-minimums parallelisation strategy.

6. Case Study Programs

To illustrate the style of JStar programs and to evaluate their performance and parallel speedup, we use four case study programs"[6]

PvWatts: This is a map-reduce style of program that reads a 192Mb CSV file generated from the PVWatts program,[7] containing hourly output measurements for solar cell installations, and calculates the average power generated during each month. Figure 4 shows the main parts of the program. The effect of the **order** declaration is to declare a causal dependency from the PvWatts tuples to the SumMonth tuples. This ensures that the last rule in Fig. 4 (triggered by SumMonth) does not run until after all PvWatts tuples have been put into the main database – if this **order** declaration was omitted then the SMT solvers would not be able to prove that that rule was stratified, so a Stratification error would be displayed.

MatrixMult: This is a naive matrix multiplication algorithm that multiplies two $N \times N$ matrices together ($N = 1000$ in this case study). The effective parallelism is that each row of the output matrix is a separate task. Each matrix multiplication is requested via a tuple, and that tuple generates one row request tuple for each output row of the matrix. Each row request tuple triggers a rule that loops over all the columns of that row, and uses a nested loop with a summation reducer to calculate the dot product results.

ShortestPath: This generates a random connected graph with one million vertices and two million edges, and then uses Dijkstra's shortest-path algorithm to find the shortest path from the starting vertex (0) to each

[6] The full source code of the JStar and Java versions of these programs is available from the JStar project page, `http://www.cs.waikato.ac.nz/research/jstar`.

[7] See `http://www.nrel.gov/rredc/pvwatts/about.html`.

```
package jstar.examples.pvwatts;
import ...
table PvWattsRequest(String filename) orderby (Req);
table PvWatts(int year, int month, int day, String hour, int power) orderby (PvWatts);
table SumMonth(int year, int month) orderby (SumMonth);
order Req < PvWatts < SumMonth;

put PvWattsRequest("large1000.csv");

foreach (PvWattsRequest req) {...code to read PvWatts tuples from *.csv...}

foreach (PvWatts pv) {put new SumMonth(pv.year, pv.month);}

foreach (SumMonth s) {
  val stats = new Statistics();
  for (record : get PvWatts(s.year, s.month)) {
    stats += record.power;
  }
  println(s.year + "/" + s.month + ": " + stats.mean)
}
```

Figure 4: A JStar program to calculate the average solar power generated in each month.

vertex. Figure 5 shows the part of the program that implements Dijkstra's shortest path algorithm. It is quite concise, because the Delta tree acts as the priority queue (ordered by the distance to the vertex), which is the main data structure used in this algorithm.

Median: This generates a relation that represents an array of 100 million random doubles and then finds the median of those values. Unlike most JStar programs, which are written in a style that is agnostic as to whether it is sequential or parallel, this program uses a more explicitly parallel algorithm. It chooses a global pivot value, divides the array into N consecutive regions, partitions each of those regions using the pivot value (similar to a Quicksort) and reports the size of those partitions back to a central controller. The controller then repeats this process (each time focusing on the partitions that must contain the median value) until only one value is left in the partition, which is the median.

We shall now compare the sequential performance of these programs, and then discuss the multicore speedup of each program in turn.

6.1. Sequential Performance of JStar versus Java

Figure 6 shows the absolute speed of the JStar case study programs when compiled with the `-sequential` flag of the JStar compiler, versus some hand-coded Java versions of the same applications. When compiling the JStar programs, we used the same JStar compiler optimisation options as will be used for the parallel versions of these programs in the following sections, and we used the same custom data structures for a few of the Gamma data structures as will be used for the parallel programs. So the sequential JStar performance shown here

```
package jstar.examples.shortestpath;
import ...
table Vertex(int index, String name) orderby(Vertex);
table Edge(int from, int to, int value) orderby(Edge);
/** Estimated shortest distance to vertex. */
table Estimate(int vertex, int distance) orderby (Int, seq distance, Estimate);
put new Estimate(0, 0);  //Set the origin.
/** Final shortest-path to each vertex. */
table Done(int vertex -> int distance) orderby (Int, seq distance, Done)
order Vertex < Edge < Int;
order Estimate < Done;
... code to generate a random graph ...

/**
 * This implements Dijkstra's shortest path algorithm.
 * The Estimate tuples are ordered by increasing distance.
 */
foreach (Estimate dist) {
  if (get uniq? Done(dist.vertex, [distance < dist.distance]) == null) {
    println("shortest path to " + dist.vertex + " is " + dist.distance);
    put new Done(dist.vertex, dist.distance);
    // process all adjacent nodes not yet done
    for (edge : get Edge(dist.vertex)) {
      if (get uniq? Done(edge.to) == null) {
        put new Estimate(edge.to, dist.distance + edge.value);
      }
    }
  }
}
```

Figure 5: JStar version of Dijkstra's Shortest Path Program.

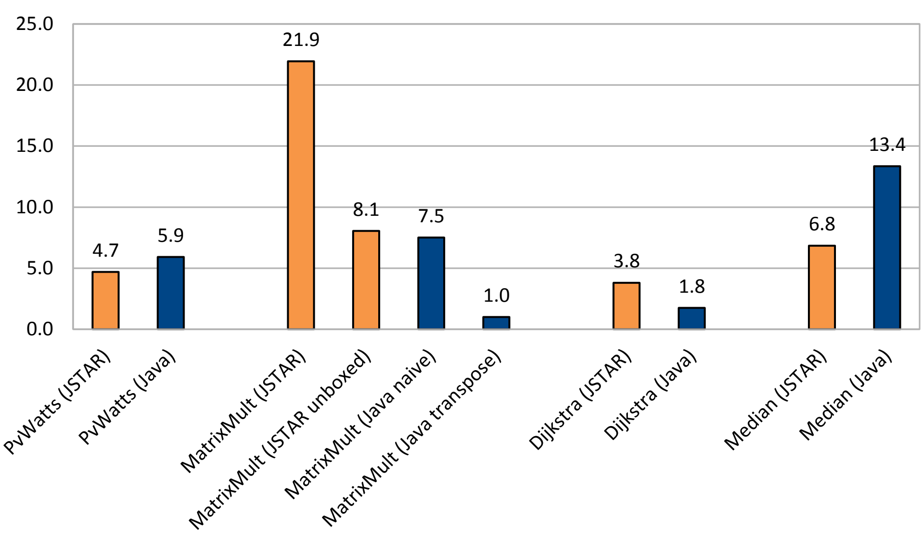

Figure 6: Absolute sequential speed of the JStar case study programs versus hand-coded Java versions, on an Intel i7-2600 CPU.

is close to the single-thread execution times of the parallel programs – the only difference is the small overhead of some Java concurrent data structures compared to their sequential equivalents (for example, ConcurrentHashMap versus HashMap).

Two of the JStar-generated programs are faster than the Java equivalents, and two are slower, so overall, this indicates that the JStar implementations are reasonably efficient. But it is worth commenting on the reasons for the differences in each case.

The JStar PvWatts program is slightly faster than the hand-coded Java version, but this is because the Java program uses the typical input reading style of BufferedReader.readline plus String.split to read the input CSV file, whereas JStar uses its own more efficient CSV library that keeps lines as byte arrays and avoids conversion to strings as much as possible.

The JStar MatrixMult program is significantly slower as generated (21.9 seconds to multiply two 1000x1000 matrices), because XText 2.3 sometimes generates code that use boxed **Integer**s rather than **int**, and it did this in the inner loop. If we manually correct this to use primitive **int**, the time comes down to 8.1 seconds, which is close to the 7.5 seconds for the naive Java matrix multiplication program. If we make an obvious improvement to the hand-coded Java version of transposing one of the matrices before multiplying them (so that the inner loop is going sequentially through *both* matrices and is more cache-friendly), then its time drops to 1.0 seconds - we could apply the same optimisation to the JStar program, but this would require modifying the JStar source code.

The JStar Dijkstra program is twice as slow as the Java version, because it pushes several million Estimate tuples through the JStar Delta tree data structures, and these are slightly less efficient than the PriorityQueue that the Java program uses.

The JStar Median program is twice as fast as the Java version, because the Java program uses Arrays.sort (a double-pivot quicksort) to find the median, whereas the JStar program uses a median-specific variant of quicksort that partitions the whole array, but then recurses only into the half of the array that contains the median.

6.2. PvWatts Speedup

A naive execution of the PvWatts program would be as follows:

1. The program starts with a request to read the input file, `large1000.csv`.
2. This tuple triggers the automatically generated read-loop rule, which uses a CSV reader library class to read the file and put all the PvWatts tuples into the Delta Set.
3. When these PvWatts tuples are moved from the Delta Set into the Gamma database, they trigger the **foreach**(PvWatts) rule in Fig. 4, which puts lots of SumMonth requests into the Delta Set. Note that JStar has a set-oriented semantics, so duplicate SumMonth tuples are discarded, and we end up with just one tuple in the Delta Set for each unique year/month combination in the input file.

4. As each of these SumMonth tuples moves from the Delta Set into the main database, it triggers the last rule in Fig. 4, which queries the PvWatts table in the main database for all tuples in that month, and uses one of the standard JStar reduce operators (Statistics) to calculate and print[8] the average power for that month.

The general purpose execution strategy outlined above is horribly inefficient for this particular application, because it requires loading the whole input file into the Delta Set and thence into the main database, before the PvWatts tuples are analyzed by the Statistics reducer. This is wasteful on both time and memory for this simple program, but if the program did more complex queries, it would be necessary to store all the PvWatts tuples before the analysis phase.

When we apply the optimisations discussed in Section 5.1 (`-noDelta=PvWatts`), and run the program on the `large1000.csv` input file (192Mb, 8,760,000 records), the sequential execution time is 23.0 seconds without the optimisation and 8.44 seconds with the optimisation.[9]

However, the PvWatts program also allows a lot of parallelism. Reading the CSV file and creating the PvWatts tuples is typically the bottleneck, but the CSV reader library can run several readers in parallel, on different parts of the input file. (Each reader continues reading a little way past the end of its region, to ensure that all records have been read. This strategy is also employed by some of the input file readers in Hadoop.[10]) All the SumMonth tuples can be processed in parallel, so that we have a separate reducer task calculating the statistics for each month. This means that we essentially get the data-flow behaviour shown in Fig. 7, with the program executing in two phases. In the first phase, N tasks read parts of the input CSV file in parallel, and put SumMonth tuples into the Delta Set and PvWatts tuples into the main database, then in the second phase we can have M tasks each processing one or more SumMonth tuple to calculate and output the statistics for one month. N and M could both default to the number of available cores, or could be chosen based on the size of the input file and the number of months. So this solution should have good scalability, though the shared data structures could become a bottleneck.

Regarding data structures, we could tell the compiler to index the year and month fields of the PvWatts table (e.g. as one hashtable) so that the query in the SumMonth rule can still be performed efficiently. The default data structure for tables in the Gamma database is a Java TreeSet for sequential code or a ConcurrentSkipListSet for parallel code, which both support ordered traversals so that queries need only traverse a subset of the table. But since

[8] As println has side effects, it is not good style to use it in rules, but we allow it for temporary debugging and tracing purposes. The kosher way of printing is to put Println tuples into the Delta Set, so that the printing side effects take place when those tuples are *removed* from the Delta Set, which follows the causality ordering. This also allows one to define an output sorting order for the Println tuples, if that is desired.

[9] For this experiment we use an Intel i7-2600 CPU, JDK 1.7.0_07 64-bit server VM with a large heap (8Gb), under Ubuntu 12-04.

[10] See `http://hadoop.apache.org`.

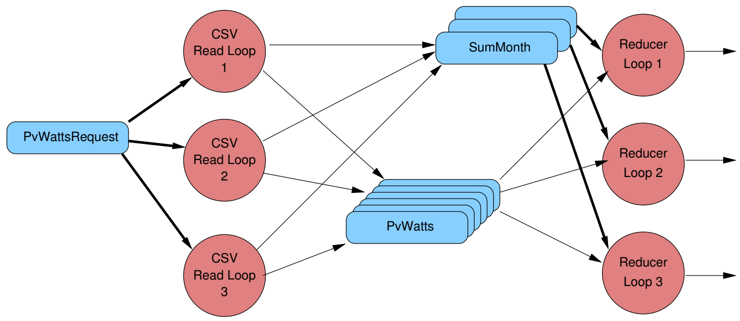

Figure 7: Two phase execution of the solar power program. Blue rectangles are tuples, and red circles are tasks executing rules – the bold arrows show the trigger tuple that starts the rule executing.

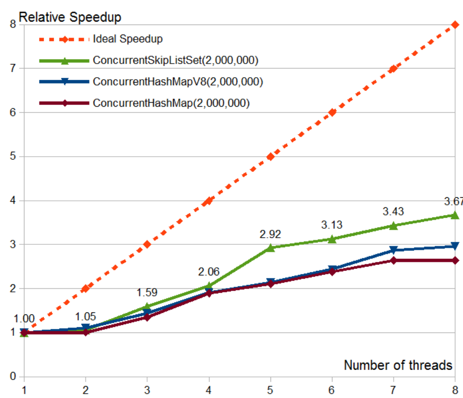

Figure 8: Relative speedup of the PvWatts program with varying fork/join pool size on a dual-CPU Intel Xeon W5590 (total of 8 cores), with alternative data structures for the PvWatts Gamma table.

this PvWatts program always queries the PvWatts table with a known year and month, we can use a HashSet or ConcurrentHashMap, which are considerably more efficient. After some experimentation, we manually implemented a custom data structure for the PvWatts Gamma database that has an array indexed by month (1..12) at the top level, and either a HashSet or ConcurrentHashMap within each entry of the array. We added this to the generated Java program by using inheritance to override one factory method. (We plan to add a compiler flag that automates the generation of these optimised 'array-of-hashsets' data structures, in the future.)

Fig. 8 shows the effect of running this optimised PvWatts program on the `large1000.csv` input file (192Mb, 8,760,000 records), varying the number of threads in the JStar Fork/Join pool (eg. `--threads=4`). Each program was run at least 20 times, the first 6 measurements (while the Hotspot compiler optimises the code) were ignored and then the average of the remaining times was taken. The *relative speedup*[11] is average, reaching nearly 4X speedup with 8 threads. The absolute speedup figures are about 35% lower, because the sequential Java data structures (eg. TreeMap) are significantly faster than the equivalent concurrent data structures (ConcurrentSkipListMap).

Given that this program inserts more than 8 million PvWatts tuples that cannot be garbage collected into the Gamma database and that we have observed up to 60% of the elapsed time being spent in the garbage collector, it is clear that garbage collection is at least partially responsible.

A more aggressive optimization would be to unfold the SumMonth rule so that its reduce loop is done incrementally as the PvWatts tuples are produced. That is, when each SumMonth tuple is generated, it immediately creates an instance of the Statistics reducer, and as the PvWatts tuples are generated they are passed to each of those reducers before being discarded. In general, this technique might require that some input tuples have to be processed by several reducers, but in this particular program, the reducer could be associated with each bucket in the PvWatts hashtable, so only a single reducer needs to be considered for each input tuple. This optimization would be more complex to apply than the other optimizations, but would eliminate the need to store the PvWatts tuples, and thus allow the program to run in a constant amount of memory, rather than proportional to the size of the input file. This optimization would not always be applicable, since more complex programs may require storing the input tuples in order to perform multiple passes over them.

6.3. PvWatts Disruptor Design

Using timing benchmarks of the various phases of the optimised PvWatts program, running in parallel mode with just 1 thread, the relative times of the various phases are:

- 16.9% reading and parsing the input file.
- 63.7% creating the PvWatts tuples and inserting them into their Gamma table;
- 3.8% creating SumMonth tuples and inserting into the Delta tree;
- 15.6% processing the SumMonth tuples by running a Statistics reducer over all the PvWatts tuples for each month.

This shows that the reader is not the main bottleneck, so we decided to investigate parallelising just the last three phases to see how much speedup can be obtained. According to Amdahl's law, the maximum speedup we could expect

[11]Relative speedup is the speedup relative to the parallel version running with one thread, while absolute speedup is relative to the fastest sequential or single-threaded parallel version.

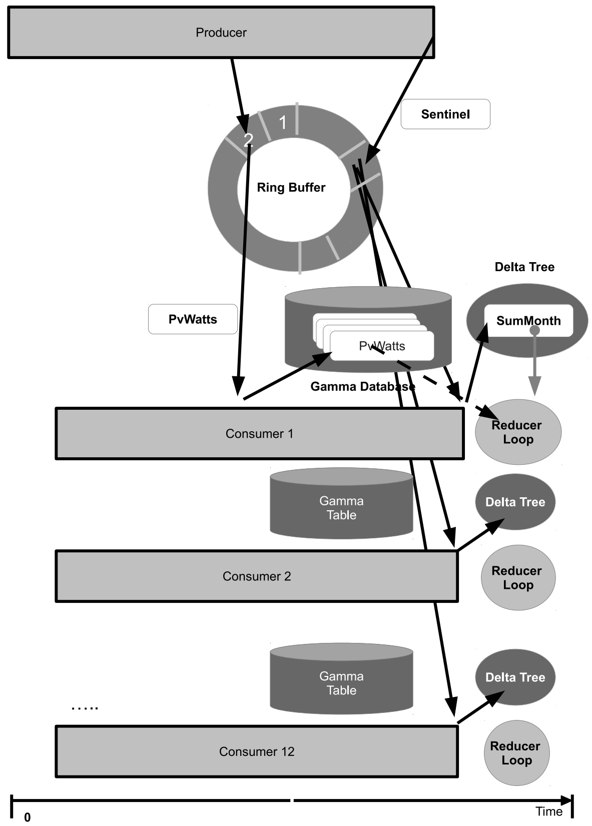

Figure 9: Workflow of the Disruptor Version of the PvWatts program. Each consumer reads just the PvWatts records for a specific month from the Producer's ring buffer, then uses local Delta and Gamma databases to process those tuples.

would be 4.2X ($= 1/(0.169 + (1 - 0.169)/12)$) with a single reader and 12 consumer processes (one for each month). To efficiently send tuples from the reader to the consumers, we use the Disruptor framework.

Disruptor[12] is a Java library developed for high-speed real-time financial exchange applications [14]. It uses a highly efficient ring-buffer to move data between producer and consumer processes, and has higher throughput, lower latency, less write contention, and a more friendly cache mechanism than other data transfer mechanisms. It is also quite flexible, with alternative implementations for single or multiple producers, single or multiple consumers, several alternative waiting strategies for consumers, and many parameters to tune the performance of the ring-buffer. It typically uses atomic CAS (Compare and

[12]Disruptor is available from `http://lmax-exchange.github.com/disruptor`.

Swap) instructions to manage access to the ring-buffer, rather than locking, and its data structures are carefully designed to reduce cache line contention and to recycle objects rather than garbage collecting them.

Our Disruptor version of PvWatts parallelizes the PvWatts program into a two-phase workflow, as shown in Fig. 9. It uses a single producer and multiple consumers to process all PvWatts tuples. The program initializes the Disruptor instance by specifying the number of consumers, the number of producers and the waiting strategy for the ringbuffer. The producer does all the CSV reading loop tasks: reading and parsing the large input file, publishing the PvWatts tuples into the ring buffer, and then sending out a sentinel tuple when the end of the input file is reached. At the same time, each consumer starts to claim PvWatts tuples from the ringbuffer. To reduce the workload of the reducer loop and improve the parallelism, we assign a separate month to each consumer. Thus, each consumer just needs to process the PvWatts tuples of one month and puts these tuples into its own Gamma database. Besides, the consumer also creates one corresponding SumMonth tuple for each PvWtts tuple and inserts this tuple into the Delta tree. When a consumer receives the sentinel tuple, it processes the SumMonth tuple from its own Delta tree, which triggers the reducer loop to query the PvWatts tuples in the Gamma table, and output their average monthly power generation.

This single-producer and multiple-consumer design removes the possibility of contention, provides good data locality, and pipelines the reducer and the consumers. If the months are evenly distributed throughout the input file, then all consumers will be busy and will progress at roughly the same rate. If there is a long sequence of records for the same month, then that consumer will have more work to do and may become a bottleneck, depending upon the size of the ring-buffer. Each consumer has its own local data storage, and puts the tuples into its local Gamma database or Delta tree.

Table 1 shows the Disruptor settings and alternatives that we used while tuning the Disruptor version of the PvWatts program. The best results with a single producer and 12 consumers were with the BlockingWaitStrategy for the consumers, a ring buffer of 1024 elements, and a producer batch size of 256.

Fig. 10 shows the execution times of the Disruptor version of PvWatts, with two different input orderings:

- the default input (unsorted) is ordered by year and month, which means that long sequences of records are processed by the same consumer. In this case, the Disruptor version with 8 threads has a speedup of 3.31 over the sequential PvWatts JStar code.

- the best case input (sorted) has better load balancing, because the input file is sorted by day of the month and time of the day, so that input records are processed by consumers in a round-robin fashion. This makes both the sequential and parallel programs faster, so the Disruptor version with 8 threads has a speedup of 2.52.

Category	**Parameter**	**Value**
RingBuffer	Event	PvWatts tuples
RingBuffer	Size of Ring Buffer	1024.
RingBuffer	Wait Strategy	BlockingWaitStrategy
RingBuffer	Claim Strategy	SingleThreaded-ClaimStrategy
Producer	Total number of Producer	1
Producer	Publish Strategy	Claim slots in a batch of 256.
Producer	Task	Read input file, create PvWatts tuples and add to ring buffer.
Consumer	Total number of Consumer	12
Consumer	Task	Process PvWatts tuples and add to Gamma.

Table 1: Disruptor Options used for PvWatts.

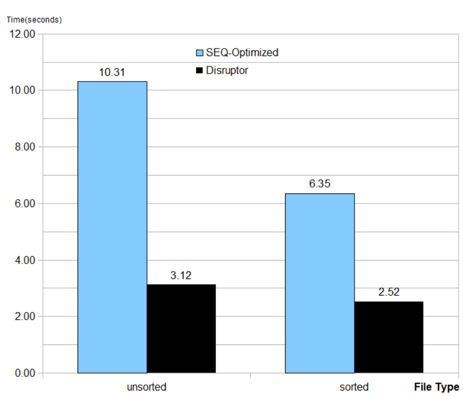

Figure 10: Execution times for the Disruptor version of PvWatts on an 8 core machine (i7-2600 with 4 cores + hyperthreading), compared to the sequential PvWatts JStar program.

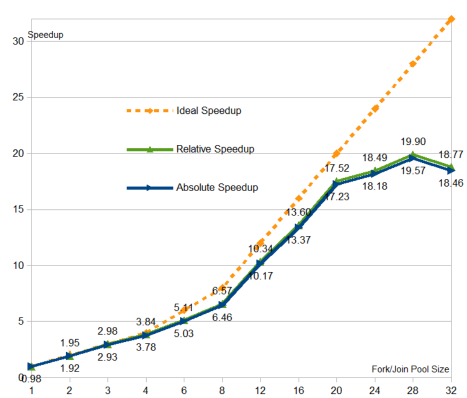

Figure 11: Speedup of the naive Matrix Multiplication program with varying fork/join pool size on a quad-CPU Intel Xeon E7-8837 (total of 32 cores).

6.4. Matrix Multiplication Speedup

Fig. 11 shows the speedup obtained by varying the thread pool size in the naive matrix multiplication program. This program is embarrassingly parallel, and has a high computation to communication ratio (after applying compiler optimisations, only one tuple per row of the output matrix needs to go through the delta set), so shows good speedup up to 20 cores. Note that to make this program more similar to standard Java matrix multiplication programs, we used a Java 2D array of integers for the gamma set of each matrix. This is an example of a commonly-useful 'native-arrays' data structure optimisation: tables that have integer keys and a single dependent value, such as:

```
table Matrix(int mat, int row, int col -> int value)
```

can be efficiently implemented using Java arrays if the keys have a limited range and are dense.

6.5. Shortest Path Speedup

The Shortest Path program effectively has two stages: first create a random graph (a tree with one million vertices, plus another one million edges between random vertices) - each edge has a random length between 1 to 10. This graph creation phase was originally done by a single rule, triggered by a command line argument tuple. But it became apparent that due to the overhead of random number generation, this phase was a bottleneck, taking more than 60% of the total time. So we modified the JStar program to allow more parallelism, by splitting the graph creation into 24 separate tasks (tuples).

Inspection of the program makes it obvious that the Estimate tuples (and the CmdLineArg tuples - not shown in Fig. 5) are the only ones that trigger rules, so we applied the `-noDelta` optimisation to the other tables, and the `-noGamma` optimisation to the Estimate table. (Some of these obvious optimisations could be applied automatically by the compiler in the future).

After these optimisations, we get the speedup results shown in Fig. 12. This has mediocre speedup, with a maximum speedup of only 4.0 (8 cores). This seems to be because the inner loop of the program puts several million `Estimate` tuples through the Delta tree, which is still not sufficiently scalable to cope with a large number of threads contending for the same branches of the tree.

6.6. Median-Finding Speedup

The main data structure in the Median-Finding program is the Data table that represents the input array containing 100 million doubles (randomly generated on each run).

```
/** The array that we are finding the median of. */
table Data(int iter, int index -> double value)
      orderby (Int, seq iter, Data, seq index);
```

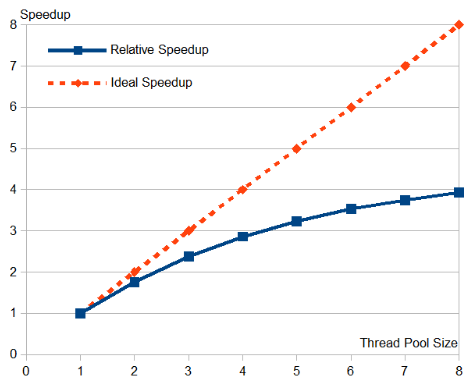

Figure 12: Speedup of the Dijkstra Shortest Path program with varying fork/join pool size on a dual-CPU Intel Xeon W5590 (total of 8 cores).

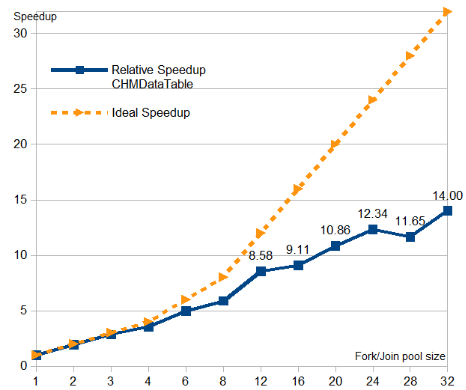

Figure 13: Speedup of the Median-Finding program with varying fork/join pool size on a quad-CPU Intel Xeon E7-8837 (total of 32 cores).

The N tasks each work on a separate area of this array, and produce an updated version of the array with each region partitioned. The `iter` field increases as these new copies of (parts of) the array are created. However, the rules only use *iter* and *iter*+1, so we only need two copies of the array. Furthermore, these Data tuples are not used as triggers, so we can use the `-noDelta` optimisation. For the Gamma implementation of the Data table, we wrote a custom subclass that stored all the values in a 2D array: `double[2][100000000]`, and used *iter* modulo 2 as the index for the outer dimension. This is the combination of the above 'native-arrays' optimisation, plus a gamma-database garbage collection optimisation that keeps only the 'current' and 'next' copies of the iterations in a table. After these optimisations, we get the speedup results shown in Fig. 13, with good speedup 8.6X up to 12 cores, and then a more gradual speedup up to a maximum of 14X with 32 cores.

7. Related Work

JStar is based on Datalog with negation, plus explicit time stamps. There have recently been several other Datalog-based language proposals [8] aimed at parallel and distributed computing. The closest to JStar is the Dedalus logic and Bloom language from Berkeley [1]. These use the same Datalog+negation semantic basis, but have a more restricted notion of timestamps than JStar, and are focussed on distributed algorithms, whereas our focus is currently on multi-core performance.

ParaSail [13] shares some design goals with JStar (eg. evaluation is parallel by default, aliasing and mutable global variables are avoided, and static analysis is used to avoid runtime errors) but it is still aimed at mutable variable programming, whereas JStar is more declarative. Another difference is the JStar goal of separating parallelism and data structure decisions from the program source – we can experiment with alternative strategies just by providing different parameters to the compiler or to the generated Java program at runtime. The Fresh Breeze project [13] also has some similar goals to JStar (determinate behaviour of fork/join parallel programs with immutable objects), but is focused on *explicit* parallelism, and hardware design of multiprocessor chips to support their execution model.

The Delite research program [4] from Stanford University's Pervasive Parallelism Laboratory (PPL) has a similar aim to JStar (writing one program and running it on many different parallel architectures), but is focussed on using *domain specific languages* (DSLs) to raise the abstraction level and capture parallel execution patterns at a high level in order to allow maximum implementation freedom. JStar is a general purpose language equivalent to Datalog+negation with a bottom-up execution semantics, so is more expressive than the current Delite DSLs. The Delite runtime can execute a task graph on parallel, heterogeneous hardware, but supports only a few parallel execution patterns such as map, reduce, zipwith etc. In contrast, the JStar runtime executes a more general directed acyclic graph of tuples, which should allow more runtime flexibility in finding ad-hoc parallelism that does not fit predetermined patterns.

JStar can be viewed as a Linda-like language, where communication between processes (rules) is done via sending and receiving tuples from a central database. However, JStar is much more declarative than a Linda system, because in JStar, tuples cannot be deleted from the tuple database, and the language is restricted to ensure that rules and programs are deterministic (except that the *order* of output tuples need not be deterministic). One Linda-like system that has very similar aims to JStar is the Intel Concurrent Collections system for C++.[14] Like JStar, it aims to abstract away from low-level communication mechanisms and communicate via tuples (objects), to specify only the semantic ordering constraints between operations rather than specifying the parallelism directly, and to allow parallelism experts much more freedom to tune the application after it is written.

8. Conclusions

We have given a brief overview of the JStar language, its compiler and its default parallelisation strategies. We have shown by example that JStar programs can exhibit a large degree of parallelism, and that it is possible to apply significant program transformations, parallelisation strategies and data structure choices purely as compiler options, *without changing the JStar source program.* On the benchmark programs we have investigated in this paper, the speedup ranges from very good (eg. for Matrix multiplication and Median) to mediocre (for Dijkstra's shortest path). In some cases, profiling showed that one rule was

[13] See `http://csg.csail.mit.edu/Users/dennis` for an overview of the Fresh Breeze project.
[14] See `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc`.

a sequential bottleneck (eg. creating the random graph in the shortest path program), so we rewrote that rule so that it could be triggered by multiple tuples, to expose more parallelism. This would be less necessary if our implementation exploited the embarrassingly parallel `for` loops within rules.[15]

We are still investigating why the speedup is not higher for the Dijkstra shortest path program (it seems to be a problem with the scalability of our Delta tree data structures). We are continuing to tune the JStar compiler and runtime to get more speed and better scalability.

An important contribution of this work is that we have demonstrated that by making the data dependencies of programs much more explicit, and by expressing the initial program using a flat relational view of the data rather than prematurely choosing particular data structures, we can:

- allow compilers to make much more aggressive transformations of programs, to introduce significant amounts of implicit parallelism and to change data structures to suit that parallelism. We do not mind whether the choice of transformation is given by the user, chosen by heuristics, or guided by performance feedback from previous executions (auto-tuning).
- allow users to visually see the possible parallelism structure in their programs, using views like those of Fig. 7.

Even more importantly, this style of programming makes it possible to try alternative parallelisation structures without changing the program's input-output behaviour, and in many cases, without changing the program source code. With our current JStar implementation, we have to write some of the optimised data structures manually, but we believe that in most cases this code could be generated automatically in the future. This separation of the application logic from the parallelisation effort enables a lightweight experimental approach to parallelisation, and could support the separation of roles between application programmers (domain experts) and parallelisation engineers (performance experts).

Acknowledgements

Thanks to James Bridgwater who developed some of the initial Java Fork/Join support for JStar, and the many other students who have worked on various aspects of JStar and Starlog. Thanks to Google for supporting part of this work via a research grant to Dr Utting.

[15]To parallelise the `for` loop that creates the random graph, it is also necessary to have support for parallel random number generators.

References

[1] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In de Moor et al. [8], pages 262–281.

[2] P. Beard. Easy parallelism. COMP520 Report, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, 2009.

[3] R. L. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 35–46. ACM, 2011.

[5] R. Clayton. *Compilation of Bottom-Up Evaluation for a Pure Logic Programming Language*. PhD thesis, Dept. of Computer Science, The Univ. of Waikato, Hamilton, NZ, 2004.

[6] J. G. Cleary, M. Utting, and R. Clayton. Datalog as a parallel general purpose programming language. Technical Report 06/2010, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, Aug. 2010. Available from `http://www.cs.waikato.ac.nz/pubs/wp`.

[7] S. Crosby. Parallelization of JStar programs on a distributed computer. Master's thesis, Department of Computer Science, The University of Waikato, 2012.

[8] O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors. *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

[9] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[10] S. McCommell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, 2004.

[11] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 193–216. Morgan Kaufmann, San Francisco, CA, 1988.

[12] D. A. Smith and M. Utting. Pseudo-naive evaluation. In *Tenth Australasian Database Conference, ADC'99, Auckland, NZ, Jan 1999*, Berlin, 1999. Springer-Verlag.

[13] S. T. Taft. Experimenting with Parasail: parallel specification and implementation language. *Ada Lett.*, 31(3):11–12, Nov. 2011.

[14] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Available from `http://lmax-exchange.github.com/disruptor`, May 2011.

[15] J. D. Ullman. Bottom-up beats top-down for Datalog. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149, New York, NY, USA, 1989. ACM.

[16] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *CACM*, 54(1):75–85, Jan. 2011.