Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

FDR into The Cloud

Thomas GIBSON-ROBINSON and A. W. ROSCOE

Department of Computer Science, University of Oxford, UK
{thomas.gibson-robinson, bill.roscoe}@cs.ox.ac.uk

Abstract. In this paper we report on a successful extension to the CSP refinement checker FDR3 that permits it to run on clusters of machines. We demonstrate that it is able to scale linearly up to clusters of 64 16-core machines (i.e. 1024 cores), achieving an overall speed-up of over 1000 relative to the sequential case. Further, this speed-up was observed both on dedicated supercomputer facilities, but more notably, also on a commodity cloud computing provider. This enhancement has enabled us to verify a system of 10^{12} states, which we believe to be the largest refinement check ever attempted by several orders of magnitude.

Keywords. CSP, FDR, model checking, MPI, supercomputer, The Cloud

Introduction

FDR (Failures Divergence Refinement) is the most widespread refinement checker for the process algebra CSP [1,2,3]. FDR takes as input a list of CSP processes, written in a lazy functional language known as machine-readable CSP (henceforth CSP_M), and is able to check if the processes refine each other according to the CSP denotational models (e.g. the traces, failures and failures-divergences models). It is also able to check for more properties (including deadlock-freedom, livelock-freedom and determinism) by constructing equivalent refinement checks.

FDR2 was released in 1996, and has been widely used both within industry and in academia for verifying systems [4,5,6]. It is also used as a verification backend for several other tools including: Casper [7] which verifies security protocols; SVA [8] which can verify simple shared-variable programs; in addition to several industrial tools (e.g. ModelWorks and ASD [9]).

Recently, FDR3 [10,11] has been released as a complete rewrite of FDR2 featuring a number of improvements including a redesigned user interface, an advanced type-checker, and improved bisimulation support [12]. Most notably, FDR3 includes a multi-core refinement-checking engine that, as the experiments in [10] demonstrate, is able to scale linearly to the number of available cores on a wide range of problems. Further, even on a single-core FDR3 is, on average, twice as fast as FDR2 [10].

In this paper, we report on an extension to FDR3 that enables it to operate on clusters of machines when checking specifications in non-divergence models (i.e. the traces and failures models). This has been extremely successful; as Section 3 explores in more detail, FDR3 was able to linearly scale to clusters of 1024 cores on not only dedicated supercomputer hardware, but more notably, also on Amazon's *pay-as-you-go* EC2 platform[1] (using up to 64, 16-core

[1] See `http://aws.amazon.com/ec2/`.

servers). Section 3 also reports on an experiment in which, using a 1024-core cluster on EC2, FDR3 was able to check a problem with 1.2 trillion states that required a total of 6TB of storage across the cluster. We were surprised that this cost only $70 to run in 5 hours on 64 16-core machines.

We believe that this represents a significant step forward for refinement-checking CSP and, more generally, for explicit-state model checking. This is primarily because it increases the maximum system size that can be checked in a reasonable time by several orders of magnitude.

Outline We begin in Section 1 by outlining the multi-core algorithm of [10] that FDR3 uses to conduct refinement checking on a single multi-core machine. Section 2 describes the extensions to the algorithm of [10] to support clusters. It also details various implementation details that are required in order to successfully scale to clusters of 64 machines and compares to related work. Section 3 reports on the experiments that we have conducted which demonstrate the ability of FDR3 to scale linearly to the number of available cores across the cluster.

1. The FDR3 Algorithm

In this section we review how the single shared-memory machine version of FDR3 functions, focusing on the parallel refinement-checking algorithm. See [10] for more details.

The core function of FDR3 is to decide if *$Spec$ is refined by $Impl$*, denoted $Spec \sqsubseteq Impl$ (all of the other properties FDR can verify, including deadlock-freedom, determinism etc. are converted into refinement checks). Informally, this requires that every *behaviour* of $Impl$ is also a behaviour of $Spec$. FDR3 has several different notions of behaviour, known as *semantic models*, that can be used to check for different properties. In this paper we consider only the *traces model* for ease; the algorithm is also applicable to the failures and failures-divergences model (the latter poses some efficiency challenges which are discussed further in Section 3). In the traces model, $Spec \sqsubseteq Impl$ is true only when every finite sequence of events that $Impl$ can perform can also be performed by $Spec$. Thus, it can be used to check simple safety properties such as *$fail$ never occurs*, or *mutex locks and unlocks must alternate*.

As input, FDR takes a list of CSP processes written in machine-readable CSP, which is a lazy functional language that has been augmented with process algebraic constructs. FDR then converts these into *labelled transition systems* (henceforth LTSs) which it subsequently operates exclusively on.

In order to decide if $Spec \sqsubseteq Impl$, FDR firstly *normalises* the $Spec$ LTS, producing a deterministic LTS that contains no τ's. Normalising large specifications is expensive, however, generally specifications are relatively small. FDR then checks if the implementation LTS refines the normalised specification LTS.

1.1. Refinement Checking

Refinement checking proceeds by performing a search over the implementation, checking that every reachable state is compatible with every state of the specification after the same trace. A breadth-first search (henceforth BFS) is performed since this produces a minimal counterexample when the check fails.

FDR3's parallel refinement checking algorithm is thus a parallel BFS algorithm that is exploring the graph induced by the product of $Spec$ and $Impl$. The FDR3 algorithm partitions the state space based on a hash function on the state pairs. Each worker is assigned a partition and has local sets to store:

```
**function** WORKER($S$, $I$, $w$)
    $done_w \leftarrow \{\}$                                   ▷ The set of states that have been visited
    $current_w \leftarrow \{\}$                                ▷ States to visit on the current ply
    $next_w \leftarrow \{\}$                                   ▷ States to visit on the next ply
    $finished_w \leftarrow true$                               ▷ True when this worker thinks it is finished
    **if** $WorkerFor(root(S), root(I)) = w$ **then**
        $current_w \leftarrow \{(root(S), root(I))\}$
        $finished_w \leftarrow false$
    **while** $\vee_{w \in Workers} \neg finished_w$ **do**
        Wait for other workers to ensure the plys start together
        $finished_w \leftarrow true$
        **for** $(s, i) \leftarrow current_w \setminus done_w$ **do**
            $finished_w \leftarrow false$
            $done_w \leftarrow done_w \cup \{(s, i)\}$
            **for** $(i', e) \in transitions(I, i)$ **do**          ▷ For each implementation transition
                **if** $e = \tau$ **then**
                    $w' \leftarrow WorkerFor(s, i')$
                    $next_{w'} \leftarrow next_{w'} \cup \{(s, i')\}$
                **else**
                    $t \leftarrow transitions(S, s, e)$
                    **if** $t = \{\}$ **then**
                        Report Trace Error                     ▷ $S$ cannot perform the event
                    **else**
                        $\{s'\} \leftarrow t$                  ▷ After normalisation, $t$ must be a singleton
                        $w' \leftarrow WorkerFor(s', i')$
                        $next_{w'} \leftarrow next_{w'} \cup \{(s', i')\}$
        Wait for other workers to finish their ply
        $current_w \leftarrow next_w$
        $next_w \leftarrow \{\}$
```

Figure 1. Each worker in a parallel refinement check executes the above function where: S is the normalised specification LTS; I is the implementation LTS; $root(X)$ returns the root of the LTS X; $transitions(X, s)$ returns the set of all (e, s') such that there is a transition $s \xrightarrow{e}_X s'$; $transitions(X, s, e)$ returns the set of s' such that $s \xrightarrow{e}_X s'$; the set of all workers is given by $Workers$. $WorkerFor(s, i)$ decides which worker should check the state pair (s, i).

- The set of state pairs in its own partition it has visited (*done*);
- The set of state pairs to visit on the current level or *ply* of the BFS ($current$);
- The set of state pairs to visit on the next level of the BFS ($next$).

When a worker visits a transition, it computes the worker who is responsible and then inserts the new state pair into the appropriate worker's $next$ set. This algorithm is presented in Figure 1.

1.2. Implementation

Whilst the abstract algorithm is straightforward, the implementation has to be carefully designed in order to obtain good performance. The main motivation behind the FDR3 implementation is to try and minimise memory usage since, on a 16-core machine, the parallel version of FDR3 can visit up to 10 billion states per hour. Given this, and the fact that we want to allow checks to make use of on-disk storage, B-Trees are a natural choice.

B-Trees The FDR3 implementation of B-Trees is relatively standard, but has been optimised to allow for efficient insertion of blocks of sorted data. The B-Tree uses a three-level caching system, as described below.

Level 1 This consists entirely of uncompressed B-Tree nodes and is normally 16MB, thus storing a few hundred B-Tree nodes.

Level 2 This is an in-memory cache of compressed B-Tree blocks. The blocks are compressed using a combination of two algorithms, the first of which takes advantage of the fact that B-Trees contain sorted data, and the second is a standard compression algorithm. This cache will gradually fill the entirety of main memory.

Level 3 If FDR3 is configured to use on-disk storage, this consists of compressed B-Tree blocks that are stored on-disk and have been evicted from the Level 2 cache.

The use of B-Trees also influences our decision to use a strict BFS, rather than a non-strict BFS (where workers are allowed to get ahead of other workers), or some other search strategy. By using BFS it makes sense for each worker w to traverse over $current_w$ in sorted order, meaning that all of the inserts into $done_w$ are in sorted order. This minimises the number of cache misses (i.e. minimises the number of accesses to B-Tree nodes not in the Level 1 cache).

Thread-Safety In order to implement the algorithm of Figure 1, we need to consider how to make it thread safe. All access to the $done$ and $current$ B-Trees is restricted to the worker who owns those B-Trees, meaning that there are no threading issues to consider. The $next$ B-Trees are more problematic: workers can generate state pairs for other workers. Thus, we need to provide some way of accessing the $next$ B-Trees of other workers in a thread-safe manner. Given the volume of data that needs to be put into $next$ (which can be an order of magnitude greater than the volume put into $done$), locking the tree is undesirable. One option would be to use fine-grained locking on the B-Tree, however this is difficult to implement efficiently.

Instead of using complex locks, FDR uses a system of buffers. Each worker w has a set of buffers, one for each other worker, and a list of buffers it has received from other workers that require insertion into this worker's $next$. When a buffer of worker w for worker $w' \neq w$ fills up, it immediately passes it to the target worker. Workers periodically check the stack of pending buffers to be flushed, and when a certain size is exceeded, they perform a bulk insert into $next$ by performing a n-way merge of all of the pending buffers to produce a single sorted buffer. This intermediate sorting also makes the actual inserts into the B-Tree more efficient.

Performance In [10] various experiments were performed that demonstrated that FDR3 was capable of linearly scaling to 16 cores on a single shared-memory machine on a wide variety of problems. Since then, we have performed additional experiments on a larger machine and have verified that FDR3 scales linearly on a single shared-memory machine with 40 cores. [10] also demonstrated that FDR3 is capable of using significant quantities on-disk storage to supplement memory, thus allowing FDR to check far larger problems than other similar tools.

2. Extending FDR3 to The Cloud

This section describes how the parallel refinement checker inside FDR3 has been extended in order to execute on clusters of machines. In Section 2.2 we also describe the theoretical network bandwidth requirements of our algorithm, whilst Section 2.3 discusses related work.

In this section each physical machine in the cluster is known as a *compute node* or, more simply, a *node*.

2.1. Implementation

The abstract algorithm that the cluster implementation uses is exactly the same as the parallel algorithm presented in Figure 1. However, it differs significantly in the implementation details, as described below.

Cluster Structure Message Passing Interface (MPI) is the most commonly used standard for writing high-performance programs that run on supercomputers, and therefore it was a natural choice to implement the cluster algorithm of FDR3. When using MPI, the first decision that has to be made is how many FDR3 processes should be run on each compute node. As in the single-machine algorithm, one *worker* (i.e. a thread executing the algorithm of Figure 1) will run on each core on each compute node. Hence, there are two obvious choices:

Single Threaded Each FDR3 process contains only one worker and there is thus one FDR3 process per core (e.g. on a 16 core node, there would be 16 single-threaded FDR3 processes running).

Multi Threaded Each FDR3 process contains one worker for *each* core on the compute node and thus only one FDR3 process runs on each physical machine (e.g. on a 16 core node, there would be one multi-threaded FDR3 process running containing 16 workers).

The single-threaded model is likely to be less efficient; no matter how well written the MPI library is, it is unlikely to be able to match the performance of the custom multi-threaded FDR3 version. This is because the multi-threaded version can take advantage of shared memory optimisations that are difficult to port to a pure message-passing system. Hence, FDR3 uses the multi-threaded model and therefore a single FDR3 process runs on each compute node.

Compute Node Structure As before, each worker uses B-Trees to store the three sets of states and, again, uses a system of buffers to transfer state pairs between different workers. As before, each worker has a buffer for each other local *worker*, and also one buffer for each remote *compute node*. In other words, writes for multiple remote workers on the same compute node are coalesced into a single outbound buffer. This is to avoid creating too many buffers as the size of the cluster increases. If each worker were to have 1024 buffers (e.g. on a 64 node cluster of 16-core machines), this would begin to require a significant quantity of memory and, in our experiments, actually slightly reduced performance.

More precisely, when a worker visits a new state pair, as before, it uses an appropriately defined $WorkerFor$ function (which is discussed further below) to decide which worker to send it to. If the destination worker is another worker w' on the same compute node then it is placed directly in a buffer for w', otherwise it is placed into the buffer for $NodeOf(w')$, where $NodeOf$ returns the compute node on which worker w' is located. When a buffer for another local worker fills up, as in the single-machine algorithm, the buffer is handed directly to the target worker, as per the single machine algorithm of Section 1.2. When a remote buffer is filled up, the worker passes it to a special thread called the *Controller*, which is discussed further below. Each worker also periodically checks for incoming buffers and, exactly as before, when a certain threshold is exceeded, performs an n-way merge and inserts them into the local $next$ tree.

In order to minimise the network bandwidth required, the blocks of state pairs for remote nodes (only) are compressed using a standard compression algorithm before being sent. This actually improves performance even when sufficient network bandwidth is available. We speculate that this may be because less data has to be copied around the MPI network implementation.

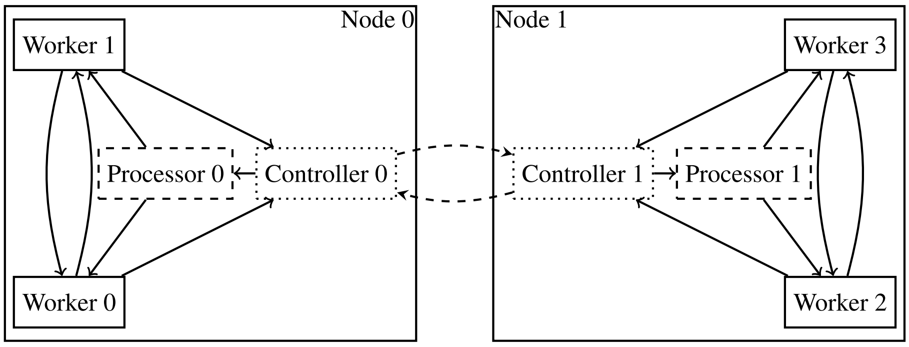


Figure 2. The architecture of the FDR3 cluster implementation, demonstrating the structure when there are two compute nodes, each with two workers and one processor. Solid lines indicate where buffers of state pairs are transferred internally within the nodes, whilst dashed lines indicate where buffers are sent via MPI.

Each node also runs one *controller* thread which is responsible for performing all communication with remote nodes[2]. The controller periodically checks for any outgoing blocks that have been sent to it by *local* workers; whenever the controller finds blocks it immediately sends them.

The controller is also responsible for receiving remote blocks generated by remote compute nodes. Since these blocks consist of state pairs for multiple local workers, the buffers must be divided up according to the destination worker. Further, the block of state pairs must first be uncompressed. Since these are computationally expensive operations that might block the controller (which has to be capable of moving 5 Gigabits of data per second, as discussed in Section 2.2) these operations are actually offloaded to a separate thread, known as the *Processor*. The number of processor threads is proportional to the number of cores on the compute node. Each processor has a state pair buffer for each local worker. The processors share a queue of state pair blocks that the controller has received and need to be processed. When a processor retrieves a block from this queue, it uncompresses the block and then inserts each state pair into the appropriate local buffer using its own buffers (sending the buffers to local workers as they fill up).

The controller also implements the global synchronisations — i.e. the waits at the start and end of the main **while** loop of Figure 1. The first of these global synchronisations uses a MPI barrier (i.e. `MPI_Barrier`), whilst the second uses a call to `MPI_Reduce` which, in addition to allowing the nodes to exchange information about how many states they have visited etc, also causes a global synchronisation.

The amount of polling that the controller does slightly affect performance. When tuning the algorithm, we found that polling around 1000 times a second achieved optimal performance, but the implementation was not sensitive to the exact polling frequency provided the polling was done at least 50 times a second.

This design is summarised in Figure 2 which shows how the different components will interact on a small two compute node cluster.

[2]By using a single thread to perform all MPI related operations, we are able to use the single-threaded MPI implementation. We did experiment with an alternative and simpler design that required a multi-threaded MPI implementation, where workers sent blocks directly to remote nodes, rather than via the controller, and the processors directly received blocks from remote nodes. However, the performance was relatively poor on large machines with more than 16 cores. In particular, a large amount of time was spent inside internal MPI locks, dramatically reducing the speed-ups observed. This was particularly noticeable when using Amazon's EC2 platform.

Note that each compute node is *over-subscribed*, in the sense that there are more threads running than there are cores (there are at least $n + 2$ threads, where n is the number of cores). Whilst this might be expected to reduce performance, our experiments revealed that reducing the number of workers to ensure that there are precisely the same number of threads as cores actually decreased performance. We believe this is because although the n worker threads are busy essentially all of the time, the m extra threads are not. Thus, if FDR were to instead use $n - m$ workers (i.e. to ensure that there is no over-subscription), this will actually leave some of the cores unused some of the time.

Work Distribution As in the single machine algorithm, the cluster algorithm used a hash function to assign work to different workers. Thus, $WorkerFor(s, i)$ was defined as $Hash(s, i)$ $\mathbf{mod}$ $WorkerCount$, where $Hash$ was a standard hash function. When we began to experiment with larger clusters consisting of 128 or more cores we noticed, for the first time, slight imbalances emerging in the distribution of work to different workers. In particular, on a particular ply we observed the difference between the earliest and latest worker finishing to be as much as 10% of the runtime of that level of the search.

In order to combat this FDR performs additional mixing on the hash function. In particular, the hash value is grouped into a larger number of buckets, and then for each bucket a random worker is assigned to work on it. Thus, $WorkerFor(s, i) \mathrel{\widehat{=}} WorkerForBucket[Hash(s, i)\ \mathbf{mod}\ (C \times WorkerCount)]$ where C is an arbitrary constant, and $WorkerForBucket$ is an array of length $C \times WorkerCount$ that returns the identifier of the worker that items in that bucket should be sent to. The $WorkerForBucket$ array is generated by randomly shuffling a vector consisting of C copies of each worker's identifier. Each worker is assigned to a particular node, meaning that $NodeOf$ can be implemented via a simple lookup. With C set to 20, this reduced the imbalance to a fraction of a percent, and also improved the speed of the single machine algorithm when running on machines with many (16 or more) cores.

This algorithm also generalises to non-homogeneous clusters (i.e. clusters where the machines are not equally powerful) by simply assigning the number of buckets proportional to the speed of the machine. FDR calculates the speed of the machine by performing a brief benchmark before starting the check. However, this process can lead to inaccurate results, due to random fluctuations in the speed of the machine. Since even a small imbalance in the work distribution can dramatically slowdown the check, the use of non-homogeneous clusters it not recommended.

2.2. Bandwidth Requirements

The most obvious potential issue with scaling the algorithm outlined in the previous section is the amount of network bandwidth required to send/receive the state pairs. In particular, in a cluster consisting of n homogeneous compute nodes, each visiting T transitions per second (i.e. generating T state pairs per second), with a state pair size of B bytes, each node will send and receive (on average) $\frac{n-1}{n} \times T \times B$ bytes per second. In fact, T is somewhat dependent on B, since larger state pairs inevitably reduce T since FDR has to do more work per state.

On a 16 core server, we have observed FDR3 visiting up to 30 million transition per second with each state pair costing 16 bytes to store. This would require 3.6 Gigabits per second to be sent and received on each compute node. As the state pairs are compressed, this tends to reduce to 2 gigabits per second in each direction.

Clearly, a commodity 1-gigabit connection is not sufficient to sustain such a volume of messages. However, a 10-gigabit ethernet connection (which are becoming increasingly common) is not only sufficient, but leaves more than enough for transient increases in rate and for future increases in processor speed or, more likely, the number of cores per compute node. On supercomputers, InfiniBand interconnects are are extremely common and provide in the range of 20 to 40-gigabit connections, which is again more than sufficient.

The above suggests that the individual network connections are sufficient, and thus it remains to consider the total volume of data that is flowing through the network. This could be problematic: in a 64 node cluster, if each machine is sending (and receiving) 2 gigabits per second, this requires the network to be able to deal with a total of 28 Gigabytes per second. Thankfully, many modern data centres use *full-bisection networks*, which allow each compute node to send and receive at the maximum rate no matter what else is occurring on the network. One common network architecture is a *fat-tree* arrangement where the network is arranged in a tree, but the links increase in bandwidth going up the tree in such a way to ensure that all nodes have sufficient bandwidth.

Thus, in practice, whilst FDR3 will make very heavy use of the network, recent developments in network design mean that FDR3 will not saturate the network. As the number of cores increases per node, this may change, but equally network bandwidth is also likely to increase (particularly on InfiniBand-based supercomputers which run simulations that are far more bandwidth intensive).

2.3. Related Techniques

The authors of [13] parallelised the FDR2 refinement checker for cluster systems that also used MPI. The algorithm they used was similar to our algorithm in that state pairs were partitioned amongst the workers and that B-Trees were used for storage. The main difference comes from the communication of $next$: in their approach this was deferred until the end of each round where a bulk exchange was done, whereas FDR3 continuously exchanges data throughout each round.

There are various trade-offs between the two approaches. Firstly, the approach of [13] introduces a large end-of-round pause which will reduce the overall speed-up that the algorithm achieves. Further, it will use more memory, since each compute node has to store all of the outgoing transitions that it finds on a given iteration, wheres under our approach we only buffer a relatively small number. On the other hand, because duplicates are eliminated before sending, the approach of [13] would require less bandwidth. However, since it requires the bandwidth to be as high as possible to reduce the overall pause, this is not as beneficial as might be hoped.

Assuming sufficient bandwidth, the time required by the approach of this paper is proportional to the number of new states generated on each ply. The time required by the approach of [13] requires time proportional to the number of new states generated on each ply (for generating the states), plus the number of unique states on each ply (for sending them). Clearly, unless there is an exceptionally large number of duplicates (which is not at all common), the approach of this paper will use significantly less time.

As a result of the above, we believe that the approach outlined in this paper has better scalability than that of [13], providing sufficient network bandwidth is available. As discussed in the previous section, this does not appear to be a problem in practice.

Several other model checkers also have support for utilising clusters of machines. LTSmin [14] allows reachability checking using clusters of machines, and achieves a near-linear speedup. DiVinE [15] implements a number of different distributed algorithms [16,17] for model-checking LTL specifications. Due to the complexity of model-checking LTL (which typically requires cycles to be found which is known to be difficult to achieve in parallel), these typically achieve reasonable, but nonetheless, sublinear speedups.

3. Experiments

In order to verify how the cluster version performed in practice, we ran a series of experiments on two different cluster systems. These experiments consisted of running various refinement checks on clusters of $1, 2, 4, \ldots, 64$ machines, thus allowing us to observe the scaling performance of the algorithm. In the case of some files we were unable to run the check on smaller numbers of machines because the smaller clusters did not have sufficient memory to complete the check; such checks are indicated using †. Table 1 details the various refinement checks, which are described in more detail below, that we attempted; they consist of a variety of checks with between 1.5 billion and 1.2 trillion states. All input files are available from the first author's webpage.

CSP Files

`bakery.n.m` is a CSP file generated by the SVA shared variable front end for FDR [8]. The shared-variable code for this, the original Lamport version of this mutual exclusion algorithm, can be found in Chapter 18 of [3]. The two parameters are the number of separate threads (between which mutual exclusion is arbitrated), and the maximum ticket value that is modelled. Each thread owns one shared and one local variable that ranges over the ticket range $\{0..m\}$, a shared boolean and a counter that ranges over $\{0..n+1\}$. Thus altogether there are n shared and n local ticket variables, n shared booleans and n local counters. Like all standard applications of SVA, this one makes significant use of FDR's state compressions applied to sub-processes, and we believe that if `bakery.6.30` could be run without this compression it would have nearly 10^{20} states. We used `bakery.6.30` as our trillion state check because we felt it more realistic than puzzles and because as the second parameter varies we found it possible to predict the overall state-space size accurately.

`bully.n` is the corrected version of the Bully Algorithm for leadership election that is described in Chapter 14 of [3]. It uses the $tock$-CSP style of writing timed processes. The parameter n is the number of nodes considered.

`cuberoll.0` is a puzzle based on rolling 8 unit cubes within a 3×3 square.

`ddb.n` is the distributed database example described in Chapter 15 of [2], in a check that considers the stabilisation of ring of n processes. This algorithm provides a way of ensuring that a group of nodes each updating a variable in a lock-free way can ensure consistency in their views of it.

`knightstour.n.m` is a simple coding of a system which explores all possible knight's tours of an $n \times m$ board, *not* using Warnsdorff's algorithm as described in Chapter 20 of [3].

`knightex.m.n` (both odd numbers) is a puzzle on an $n \times m$ board in which the centre square is empty, and the rest divided into equal-sized white and red regions. The objective of the puzzle is to swap the white and red pegs. Given $K = n \times m$, it has $K \times \binom{K-1}{(K-1)/2}$ states.

`solitaire.0` is the standard FDR benchmark of a CSP coding of the peg solitaire puzzle as described in Chapter 15 of [2]. This has 187M states and is too small to include in our experiments, but the 1024-core cluster completed it in 2.2 seconds. We note that when [2] was published this example was considered too large for FDR. For these experiments we instead used larger versions, where `solitaire.n` has a group of 3 extra pegs added to n of the four legs of the cross (giving $33 + n \times 3$ slots in all).

Table 1. The different refinement checks used in the experiments.

Input File	States (10^9)	Transitions (10^9)	Memory (GB)
`bakery.6.8`	21.3	119.1	102
`bakery.6.30`	1174.7	6585.0	6100
`bully.8`	5.9	66.9	33
`cuberoll.0`	7.5	20.1	35
`ddb.10`	51.1	352.0	406
`knightex.3.11`	19.8	67.3	154
`knightex.5.7`	81.2	355	632
`knightstour.5.9`	123.9	207.3	574
`solitaire.1`	1.6	14.0	7
`solitaire.2`	11.6	113.8	50

Cluster Systems The first cluster system we experimented with was *Arcus*, which is 1472 core supercomputer at the University of Oxford. It consists of 92 compute nodes, each of which has two 8-core 2GHz Intel Xeons and 60GB of RAM. The compute nodes are connected using a 40Gbit/s InfiniBand interconnect, arranged in a fat-tree architecture. On *Arcus* FDR3 used MVAPICH 1.8 as the MPI implementation. Detailed experimental results are given in Figure 4.

More interestingly, we also utilised commodity cloud computing hardware by renting time on Amazon's Elastic Compute Cloud (EC2). This allows you to rent machines of varying size by the hour and on-demand. On EC2 we utilised clusters of up to 64 `r3.8xlarge` machines, each of which had two 8-core 2.6GHz Intel Xeons and 240GB of RAM. The machines are connected using a 10-gigabit network and were configured in a *cluster placement group*, which Amazon claims ensures low-jitter full-bisection network bandwidth. On EC2 FDR3 used MPICH 3.1 as the MPI implementation. Detailed experimental results are given in Figure 3.

Scaling Performance As shown in Figure 3b, on EC2 FDR3 achieves an average speedup of 67 over a single server on a 64 machine cluster, which equates to a speedup of over 1000 over the sequential version. Surprisingly, this is a *super-linear* speedup. We believe that this is because the size of the B-Trees decreases as the cluster size increases, meaning that the Level 1 cache of B-Tree blocks is more likely to be useful (i.e. the hit rate increases).

Figure 3b and Figure 4b also indicate that the cluster version imposes a small overhead since the average speedup from one to two nodes is 1.61. Some of this slow down will be because the state pairs blocks have to be compressed before being sent to remote nodes, but the source of the remainder is unclear to us. Profiling indicates that using MPI causes memory bandwidth (i.e. the amount of data transferred between the RAM and the processor's cache) to increase, and since FDR3 is already memory-bound, this appears to slowdown FDR3 slightly. Thanks to the superlinear scaling observed above, this effect is cancelled out with clusters of 32 compute nodes or more.

Some files also appear to scale better than others. For example, Figure 3a and Figure 4a indicate that whilst `knightex.3.11` and `bully.8` achieved superlinear scaling (on EC2, 65 and 73 respectively), `bakery.6.8` and `solitaire.2` achieved only a sublinear scaling (on EC2, 54 and 60 respectively). This is because of the strucutre of the problem; generally, problems that have a high number of states per ply will scale better because there are fewer global barrier synchronisations required. This effect is also observed when scaling FDR3 on a single machine, as shown in the experiments of [10].

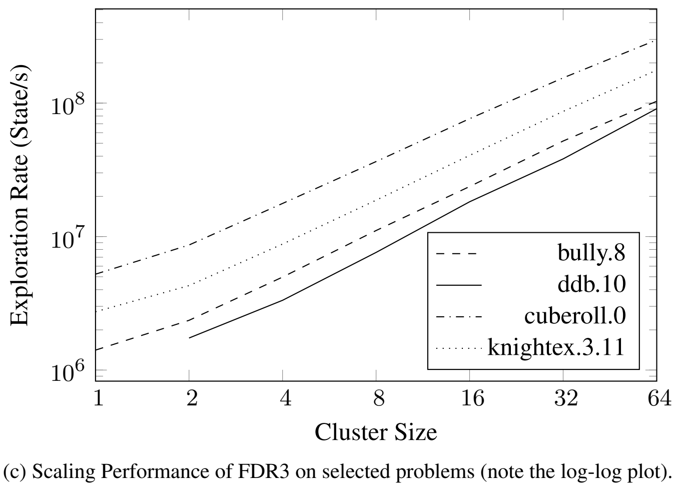
Other than the fact that each compute node on EC2 is faster than those on *Arcus* (since *Arcus* dates from 2010), there is little difference in speed between the two platforms. This is

Input File	Time by Cluster Size (s)						
	1	2	4	8	16	32	64
`bakery.6.8`	10069	5919	2940	1378	675	356	186
`bakery.6.30`	†	†	†	†	†	37701	20680
`bully.8`	4183	2502	1186	532	249	113	57
`cuberoll.0`	1439	870	425	206	98	49	25
`ddb.10`	†	29389	15329	6711	2801	1337	562
`knightex.3.11`	7259	4622	2257	1059	489	229	112
`knightex.5.7`	†	†	15987	7195	2954	1453	586
`knightstour.5.9`	†	†	12165	5926	2201	1091	441
`solitaire.1`	568	356	174	99	52	21	12
`solitaire.2`	5383	3726	1982	920	439	190	90

(a) Absolute amount of time taken by each refinement check.

Input File	Speedup Factor by Cluster Size						
	1	2	4	8	16	32	64
`bakery.6.8`	—	1.70	2.01	2.13	2.04	1.09	1.91
`bully.8`	—	1.67	2.11	2.23	2.14	2.19	1.99
`cuberoll.0`	—	1.65	2.05	2.06	2.10	2.02	1.92
`ddb.10`	†	—	1.92	2.28	2.40	2.10	2.38
`knightex.3.11`	—	1.57	2.05	2.13	2.17	2.14	2.04
`knightex.5.7`	†	†	—	2.22	2.44	2.03	2.48
`knightstour.5.9`	†	†	—	2.05	2.69	2.02	2.48
`solitaire.1`	—	1.59	2.05	1.76	1.89	2.48	1.80
`solitaire.2`	—	1.44	1.88	2.16	2.09	2.31	2.11
Average	—	1.61	2.01	2.11	2.22	2.13	2.09
Average vs 1 Node	—	1.61	3.23	6.82	15.12	32.23	67.43

(b) Speedup figures for the experiments run on EC2. In the above grid, except for the average row, an entry in a column corresponding to n compute nodes refers to how much faster FDR3 was with n nodes than with $\frac{n}{2}$ nodes (i.e. a value of 2 would indicate linear scaling, > 2 indicates superlinear scaling, and < 2 indicates sublinear scaling).



(c) Scaling Performance of FDR3 on selected problems (note the log-log plot).

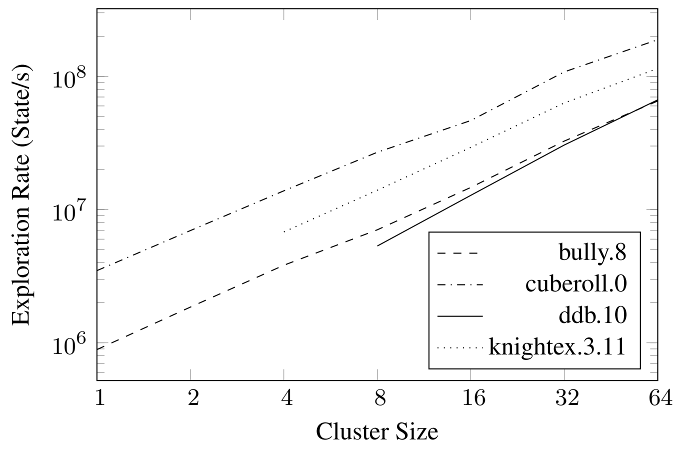
Figure 3. Results for the experiments run on EC2.

Input File	Time by Cluster Size (s)						
	1	2	4	8	16	32	64
`bakery.6.8`	†	6974	3246	1717	875	454	246
`bully.8`	6629	3181	1543	836	402	181	90
`cuberoll.0`	2160	1081	544	279	161	70	40
`ddb.10`	†	†	†	9578	3990	1672	766
`knightex.3.11`	†	†	2918	1419	677	314	173
`knightex.5.7`	†	†	†	†	4169	1692	773
`knightstour.5.9`	†	†	†	†	3315	1285	569
`solitaire.1`	707	424	211	115	61	34	20
`solitaire.2`	†	4610	2506	1140	527	262	149

(a) Absolute amount of time taken by each refinement check.

Input File	Cluster Size						
	1	2	4	8	16	32	64
`bakery.6.8`	†	—	2.15	1.89	1.96	1.93	1.85
`bully.8`	—	2.08	2.06	1.84	2.08	2.23	2.00
`cuberoll.0`	—	2.00	1.99	1.95	1.73	2.32	1.75
`ddb.10`	†	†	†	—	2.40	2.39	2.18
`knightex.3.11`	†	†	—	2.06	2.10	2.16	1.81
`knightex.5.7`	†	†	†	†	—	2.46	2.19
`knightstour.5.9`	†	†	†	†	—	2.58	2.26
`solitaire.1`	—	1.67	2.01	1.83	1.89	1.79	1.68
`solitaire.2`	†	—	1.84	2.20	2.16	2.01	1.76
Average	—	1.92	2.01	1.96	2.05	2.21	1.94
Average vs 1 Node	—	1.92	3.85	7.55	15.45	34.10	66.19

(b) Speedup figures for the experiments run on *Arcus*. The entries in the above grid are as per Figure 3b.



(c) Scaling Performance of FDR3 on selected problems (note the log-log plot).

Figure 4. Results for the experiments run on *Arcus*.

Table 2. The memory usage of FDR3 on EC2 on a selection of problems.

Input File	Memory Usage by Cluster Size (GB)						
	1	2	4	8	16	32	64
`bakery.6.8`	80	83	87	91	96	101	108
`bully.8`	31	31	33	34	36	36	41
`cuberoll.0`	29	30	31	33	35	39	42
`solitaire.2`	42	43	45	47	49	53	57

an encouraging sign given that access to specialised supercomputers such as *Arcus* is often hard to obtain, whereas access to EC2 is available to anyone.

10^{12} *States and Beyond* In order to probe the capabilities of the cluster version we ran a refinement check on a large version of the bakery algorithm. This check had 1.2 trillion states over 1,200 plys, 6 trillion transitions, and required 6TB of storage across the cluster. This file also uses *compression*, whereby a LTS is converted into a smaller equivalent LTS (as many large CSP scripts do). Thus, the check of 10^{12} compressed states actually corresponds to checking around 10^{20} uncompressed states.

This check could not be executed on *Arcus*, since it had insufficient memory, but it could be on EC2 on both a 32 and 64-node cluster, taking 10.5 hours and 5.7 hours, respectively. The sublinear speedup is likely to be due to the fact that this problem has a very large number of plys (around 1300), meaning that a large number of the plys are relatively small. As discussed above, the cluster version is more efficient on problems that have a smaller number of larger plys.

Network Bandwidth During the experiments on EC2, the utilised network bandwidth was also monitored (unfortunately, the relevant data was not accessible on *Arcus*). The network bandwidth was highly problem dependent (since it depends on the average out-degree of a state pair), and the highest rates were observed when verifying `solitaire.1`, with each node sending/receiving on average around 2.4 Gbit/s, with occasional spikes of up to 6 Gbit/s (near the end of the plys). This is only a little over the estimates of Section 2.2, possibly because the MPI implementation has to send additional data. Note that since network bandwidth was not an issue, no attempt was made to optimise FDR3's network usage.

Memory Usage Table 2 shows how the memory usage varies as the cluster size increases. This table clearly shows that FDR3 is consuming up to 20% more memory as the cluster size increases. There are two primary reasons for this. Firstly, each compute node has its own local cache and thus, as the size of the cluster increases, the total amount of storage used by the cache increases. The second reason is more subtle. As explained in Section 1, FDR3 compresses B-Tree blocks by exploiting the fact that the keys are sorted, and thus only stores the differences between the keys. However, the hash function that is used does not respect this ordering, meaning that consecutive keys may be assigned to different B-Trees, thus reducing the potential for compression.

Conclusions

In this paper we have presented a cluster version of the FDR3 refinement checker. We have demonstrated that it is able to linearly scale up to at least clusters of 64 16-core servers on a wide variety of traces and failures refinement checks, achieving an overall speed-up of over 1000 versus the sequential version. This represents an enormous increase in the size of problem that can be realistically checked by FDR in a reasonable period of time. This is exemplified by the fact that we were able to successfully complete a refinement check with

10^{12} states, which is at least an order of magnitude more than the single machine version of FDR3 could manage, and at least three orders of magnitude more than FDR2 could manage in a reasonable period of time.

The fact that FDR3 is capable of operating on commodity cloud computing services is particularly interesting. Since Amazon requires nothing more than a credit card to register, it is possible for anyone to rent a cluster of 64 powerful servers and then use FDR3 to check extremely large problems. Further, EC2 is relatively inexpensive: in total including some development time and and the time required to run all of the experiments, we spent only \$400 on EC2 at an average cost of \$0.26 per server per hour[3].

As noted in Section 2.3, whilst there have been attempts to scale other model checkers to work efficiently on clusters, the use of temporal logics has hindered their ability to take full advantage. This is one benefit of FDR: as the traces and failures models do not require cycles to be found, and can check a very large and useful class of specifications, FDR is able to use a vastly more efficient parallel implementation. However, it should be noted that parallelising checks in the failures-divergences model is likely to be challenging since checking for divergences does require cycles to be found. Efficiently parallelising checks in the failures-divergences model is the subject of continuing research.

Future Work Presently, we statically partition the graph based on a hash function. Whilst this produces (essentially) perfect balancing, it does cause an increase in memory usage, as discussed in Section 3. We intend to attempt to eliminate this by using alternative methods of assigning work that attempt to group similar keys together. This may require us to investigate dynamic partitions, which would additionally also allow FDR3 to cope with nodes that started to slow down (relative to another node) for some unknown reason (perhaps due to a malfunction, or otherwise).

As discussed in Section 2.2, FDR3 requires relatively significant quantities of bandwidth in order to exchange states. Whilst this does not pose any challenge to hardware at the moment, if the number of cores on a given server increases faster than the bandwidth available, then this may become a problem. Therefore, we would like to investigate if it is possible to assign state pairs to workers in such a way to increase the likelihood that successor states will also be assigned to the same worker. This would reduce the total amount of network bandwidth required.

As demonstrated in Section 3, there appears to be a slowdown with simply enabling the MPI based solution. It may be possible to reduce, or even eliminate this by moving to a custom non-MPI based implementation for the exchange of node pairs.

Availability

FDR3 is available for Linux and Mac OS X from `https://www.cs.ox.ac.uk/projects/fdr/`. The cluster version is included in the above release only for Linux. FDR3 is free for academic and research purposes whilst commercial licensing is available.

Acknowledgements

We are grateful to Philip Armstrong and Gavin Lowe for many interesting discussions concerning this work. We would also like to thank the anonymous reviewers for many helpful

[3]This is largely thanks to Amazon's *spot pricing* whereby they essentially auction spare compute capacity. The spot price can be variable, but was extremely stable at \$0.26 per hour on the type of server we were interested in.

comments. The authors would like to acknowledge the use of the Advanced Research Computing (ARC) in carrying out this work. Research into FDR3 has been partially sponsored by DARPA under agreement number FA8750-12-2-0247.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[2] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[3] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[4] Jonathan Lawrence. Practical Application of CSP and FDR to Software Design. In *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *LNCS*. 2005.

[5] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40(1), 2001.

[6] Clemens Fischer and Heike Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In *IFM'99*. Springer, 1999.

[7] Gavin Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1-2), 1998.

[8] A. W. Roscoe and David Hopkins. SVA, a Tool for Analysing Shared-Variable Programs. In *Proceedings of AVoCS 2007*, 2007.

[9] Philippa J Hopcroft and Guy H Broadfoot. Combining the box structure development method and CSP for software development. *Electronic Notes in Theoretical Computer Science*, 128(6):127–144, 2005.

[10] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Model Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[11] University of Oxford. *Failures-Divergence Refinement—FDR 3 User Manual*, 2014. `https://www.cs.ox.ac.uk/projects/fdr/manual/`.

[12] Alexandre Boulgakov, Thomas Gibson-Robinson, and A.W. Roscoe. Computing Maximal Bisimulations. In *ICFEM*, 2014.

[13] Michael Goldsmith and Jeremy Martin. The parallelisation of FDR. In *Proceedings of the Workshop on Parallel and Distributed Model Checking*, 2002.

[14] Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NASA Formal Methods*, volume 6617 of *LNCS*. 2011.

[15] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, volume 8044 of *LNCS*, 2013.

[16] Jiri Barnat, Lubos Brim, and Pavel Simecek. Cluster-Based I/O-Efficient LTL Model Checking. In *ASE*, pages 635–639. IEEE Computer Society, 2009.

[17] Kees Verstoep, Henri E. Bal, Jiri Barnat, and Lubos Brim. Efficient Large-Scale Model Checking. In *IPDPS*, pages 1–12. IEEE, 2009.