# Concurrent Computation of Bisimilarity Distances

Qiyi Tang

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

**Abstract.** Bisimilarity distances capture the similarity of states of labelled Markov chains. There are several ways to compute these distances. The algorithm we are interested in is by Bacci et al. They proposed an on-the-fly algorithm in which several optimizations are applied. We implement the algorithm without these optimizations and parallelize it. We test the correctness of the implementations by generating thousands of random labelled Markov chains and use our algorithms to compute the bisimilarity distances. We also test the performance of the sequential and concurrent implementations by recording the time spent on computing the bisimilarity distances of these Markov chains. The performance test shows that the sequential implementation is much faster than the concurrent implementation.

## 1 Introduction

Markov chains or discrete time Markov chains can model many real life probabilistic processes. We give the formal definition of Markov chain (Definition 1) later. In order to do state space reduction it is sometimes useful to know whether two states are similar or not. Probabilistic bisimulation, an extension of bisimulation, is a binary process relation introduced by Larsen and Skou [1]. If two Markov chains are bisimilar, they should match each other's moves. Also, the probabilities associated with the moves should be the same.

Desharnais et al., however, mentioned that in real-world applications two Markov chains might be very similar but are not bisimilar ([2]). Therefore, inspired by the Kantorovich metric, Desharnais et al. proposed a pseudo-metric to measure the distances of labelled Markov chains ([2]). We call these distances bisimilarity distances, as done in the paper of Bacci et al. [3]. While there are several ways to compute the bisimilarity distance, for example, van Breugel and Worrell compute it using linear programming ([4]), we present the one which is proposed by Bacci et al. [3].

We start with basic concepts namely Markov chain, transportation problem and coupling. With these concepts explained, we are able to describe the computation of bisimilarity distances. The sequential algorithm is shown in the next

section. Based on the sequential algorithm, ideas of making the algorithm concurrent are discussed. We implement a concurrent version of the algorithm which passes the correctness test. Finally, we focus on testing the performance. The performance of the sequential and concurrent implementations are evaluated on a machine in the Intel®Manycore Testing lab (MTL)[1].

## 2 Computation of bisimilarity distances

In this section, we give the definition of labelled Markov chains and introduce the transportation problem first. Then we will characterize the notion of bisimilarity distances.

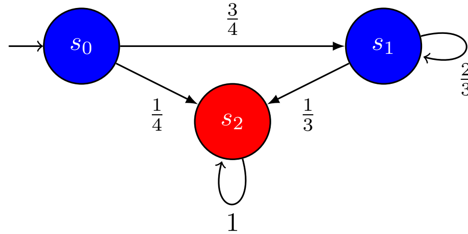
**Definition 1** *A* labelled Markov Chain *is a tuple* $(S, A, \mathbf{P}, l)$ *where*

- $S$ *is a finite, non-empty set of* states*,*
- $A$ *is a finite, non-empty set of* labels*,*
- $\mathbf{P} : S \times S \to [0, 1] \cap \mathbb{Q}$ *is the* transition probability function *such that for all states* $s \in S$*,*

$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$

- $l : S \to A$ *is the* labelling function*.*

**Example 1** *The Markov chain depicted by*




*has three states and five transitions where state* $s_0$ *is the initial state. The transition probability function can be easily extracted from the above diagram. For example,* $\mathbf{P}(s_0, s_1) = \frac{3}{4}$ *and* $\mathbf{P}(s_2, s_2) = 1$*. The set of labels is* $\{blue, red\}$*. As can be seen in the Markov chain,* $s_0$ *and* $s_1$ *are labelled blue and* $s_2$ *is labelled red.*

Let's formalize balanced transportation problem as it is used in the algorithm of computing the bisimilarity distance. The problem is to minimise the total cost of shipping from a number of sources to a number of sinks. A balanced transportation problem TP($c$, $s$, $d$) consists of

[1] The machine in the Intel®Manycore lab has 4 CPUs of Xeon®E7-4860 @ 2.27GHz. There are 80 cores in total.

- $n$ sources and a supply function $s : [1..n] \to \mathbb{R}_0^+$, $s(i)$ denotes the amount of units $source_i$ supplies
- $m$ sinks and a demand function $d : [1..m] \to \mathbb{R}_0^+$, $d(i)$ denotes the amount of units $sink_i$ demands
- A cost function $c : [1..n] \times [1..m] \to \mathbb{R}_0^+$, $c(i, j)$ ($i \in [1..n]$, $j \in [1..m]$) denotes the cost of transporting one unit from $source_i$ to $sink_j$.

A balanced transportation problem requires the total supplies to match the total demands, that is, $\sum_{i\in[1..n]} s(i) = \sum_{i\in[1..m]} d(i)$. The problem is to find a shipping function $x : [1..n] \times [1..m] \to \mathbb{R}_0^+$ such that

- The shipping function and the supplies function satisfy the following equation,

$$\forall i \in [1..n], \sum_{j\in[1..m]} x(i, j) = s(i) \tag{1}$$

- The shipping function and the demands function satisfy the following equation,

$$\forall j \in [1..m], \sum_{i\in[1..n]} x(i, j) = d(j) \tag{2}$$

- $x = \arg\min_x \sum_{i\in[1..n],j\in[1..m]} x(i, j) \cdot c(i, j)$, which means the shipping function $x$ leads to the minimum total cost.

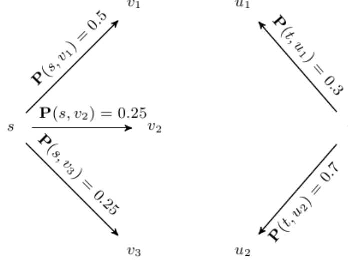



Fig. 1: A balanced transportation problem $TP(c, s, d)$

**Example 2** *The following is a balanced transportation problem. Figure 1 is a Markov chain $(S, A, \mathbf{P}, l)$. The set of states $\{v_1, v_2, v_3\}$ acts as the sources while the set of states $\{u_1, u_2\}$ acts as the sinks. The transition function $\mathbf{P}$ specifies the supplies and demands, e.g. the supply of the source $v_1$ is $\mathbf{P}(s, v_1)$ and the demand of the sink $u_2$ is $\mathbf{P}(t, u_2)$. The cost function $c$ is shown in the table below.*

| *Cost Function* | | |
|---|---|---|
| | $u_1$ | $u_2$ |
| $v_1$ | 0.6 | 0.4 |
| $v_2$ | 0.2 | 0.8 |
| $v_3$ | 0.9 | 0.1 |

*A possible shipping function is illustrated in Figure 2. The total cost in this case is* $0.3 \times 0.6 + 0.2 \times 0.4 + 0.25 \times 0.8 + 0.25 \times 0.1 = 0.485$. *By applying one of the algorithms (e.g. Chapter 8 of [5]) which checks for the optimality, this cost is found not to be minimal so that the shipping function is not optimal.*

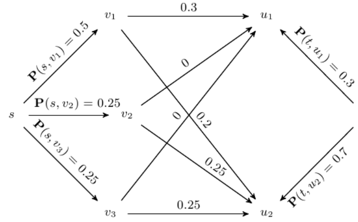



Fig. 2: A shipping function for $TP(c, s, d)$

According to Chen et al. [6], the bisimilarity distance is characterized based on the notion of *coupling*, the definition of *coupling* is given in the following.

**Definition 2** *A* coupling *for* $\mathcal{M} = (S, A, \mathbf{P}, l)$ *is a Markov chain* $\mathcal{C} = (S \times S, A \times A, \omega, \mathbf{l})$ *where*

- $\omega \in S \times S \to [0, 1]$ *s.t.* $\forall s, t \in S, \omega((s, t), \cdot) \in \mathbf{P}(s, \cdot) \otimes \mathbf{P}(t, \cdot)$ *which means* $\forall s, t \in S$,

$$\forall v \in S. \sum_{u \in S} \omega((s, t), (u, v)) = \mathbf{P}(s, v) \tag{3}$$

$$\forall u \in S. \sum_{v \in S} \omega((s, t), (u, v)) = \mathbf{P}(t, u) \tag{4}$$

- $\mathbf{l} : S \times S \to A \times A$ *and* $\forall s, t \in S.\ \mathbf{l}(s, t) = (l(s), l(t))$

Given a coupling $\mathcal{C} = (S \times S, A \times A, \omega, \mathbf{l})$ and $\lambda \in (0, 1]$, the operator $\Gamma_\lambda^{\mathcal{C}} : [0, 1]^{S \times S} \to [0, 1]^{S \times S}$, for $d : S \times S \to [0, 1]$ and $s, t \in S$, is defined by:

$$\Gamma_\lambda^{\mathcal{C}}(d)(s, t) = \begin{cases} 1 & if \quad l(s) \neq l(t) \\ \lambda \cdot \sum_{u,v \in S} d(u, v) \cdot \omega((s, t), (u, v)) & if \quad l(s) = l(t) \end{cases} \tag{5}$$

**Definition 3** *Given a Markov chain* $\mathcal{M}$ *and a discount factor* $\lambda \in (0, 1]$, *the* $\lambda$-*discounted* bisimilarity distance $\Delta_\lambda^{\mathcal{M}}$ *for* $\mathcal{M}$ *is* $\Delta_\lambda^{\mathcal{M}} = min \left\{ \gamma_\lambda^{\mathcal{C}} \mid \mathcal{C} \text{ is a coupling for } \mathcal{M} \right\}$ *where* $\gamma_\lambda^{\mathcal{C}}$ *is the least fixed point of* $\Gamma_\lambda^{\mathcal{C}}$.

Note that the Equation 3 and Equation 4 are very similar to Equation 1 and Equation 2. Therefore according to Definition 3, the computation of bisimilarity distances boils down to solving the transportation problem $TP(d, \pi(s, \cdot), \pi(t, \cdot))$ for every pair of states $(s, t) \in S \times S$.

## 3 Sequential algorithm

According to Definition 3, the sequential algorithm is to search for a coupling which gives the minimum value of $\gamma_\lambda^{\mathcal{C}}$. In the paper of Bacci et al. [3] an optimized on-the-fly algorithm is implemented. Instead of presenting that algorithm, we give the pseudocode of the algorithm without the optimizations mentioned in the paper. The correctness of this unoptimized algorithm will be proved in the future.

Given a labelled Markov chain $\mathcal{M} = (S, A, \mathbf{P}, l)$ and a discount factor $\lambda$, the goal is to compute the $\lambda$-discounted bisimilarity distances for every pair of states in the Markov chain. We use a matrix to store these bisimilarity distances. The algorithm is divided into two phases. The first phase involves a series of initializations. If a pair of states have different labels, we set the distance for this pair to be 1; if two states are the same, we set the distance to be 0. For all the other pairs, where the two states are different but have the same label, we assign arbitrary couplings. Based on the initial couplings, the distances of the states are then computed.

Coupling can also be defined in terms of a transportation problem. The coupling for a pair of states $(s, t)$ is the shipping function of the transportation problem $TP(d, \mathbf{P}(s, \cdot), \mathbf{P}(t, \cdot))$, where $\mathbf{P}(s, \cdot)$ is the set of supplies and $\mathbf{P}(t, \cdot)$ is the set of demands, $d$ is the cost matrix and in this algorithm, it is the matrix of bisimilarity distances.

The second phase is a loop and the stop criterion is all the couplings are optimal. In each iteration, we pick a pair of states for which the coupling is not optimal, we then replace the non-optimal coupling with the optimal one. We recalculate the distances with the updated coupling and then check if the stop criterion is satisfied.

The algorithm is presented in Algorithm 7 in the appendix. There are two inputs to this algorithm: a Markov chain and a discount factor. $\Omega$ is a coupling and it contains for every state pairs $(s, t)$ a coupling in $P(s, \cdot) \otimes P(t, \cdot)$. The set $G$ includes all pair of states where the two states have different labels.

The first part (line $1 - 15$) serves as initialization of the algorithm. We set distances to be 1 if the pair of the states have different labels and 0 if the pair of states are the same. We arbitrarily initialize $\Omega$ for the pair of states which are not in the set $G$. Then Algorithm 6 is called to calculate $\gamma_\lambda^{\mathcal{C}}$ given the current couplings.

The second part (line $16 - 20$) of Algorithm 7 is a loop. The stop criterion is when all the couplings are optimal. It iteratively updates $\Omega[(s, t)]$ while it is not optimal for the pair of states $(s, t)$. After assigning the optimal schedule $TP(d, \mathbf{P}(s, \cdot), \mathbf{P}(t, \cdot))$ to $\Omega[(s, t)]$, $d$ is recomputed and updated.

Algorithm 6 is a subroutine which calculates $\gamma_\lambda^{\mathcal{C}}$ given the current coupling. To compute $\gamma_\lambda^{\mathcal{C}}$, the least fixed point of Equation 5, is equivalent to computing the reachability probability of every pair of states $(s,t)$ in $\Omega$ to $\{(s,t) \in S \times S \mid l(s) \neq l(t)\}$. In this case, the coupling is treated as a normal labelled Markov chain. We can compute the above-mentioned reachability problem by solving the following linear equation: $\mathbf{x} = \lambda(\mathbf{A} \cdot \mathbf{x} + \mathbf{b})$, where $\mathbf{x} = (x_{(s,t)})_{(s,t)\in G'}$, $\mathbf{A} = (\Omega[(s,t)](u,v))_{(s,t),(u,v)\in G'}$, $\mathbf{b} = (\sum_{(u,v)\in G} \Omega[(s,t)](u,v))_{(s,t)\in G'}$ and $G = \{(s,t) \in S \times S \mid l(s) \neq l(t)\}$, $G' = (S \times S) \backslash G$.

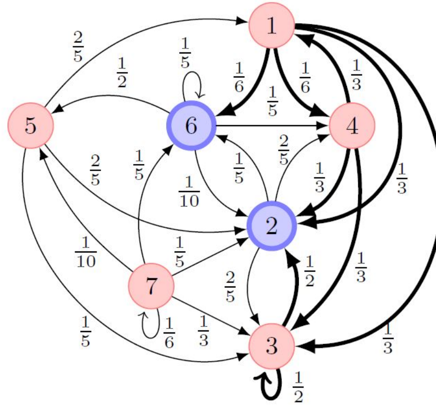



Fig. 3: A Markov chain. *Adapted from [3].*

**Example 3** *We present an example which is adapted from the paper by Bacci et al. ([3]). The Markov chain shown in Figure 3 and $\lambda = 1$ are the two inputs of Algorithm 7. For every pair of states $(s,t) \notin G$, we assign an arbitrary coupling for that pair. For example, the couplings for the pair $(1,4)$ and $(3,4)$ are initialized according to the first two tables in Table 1.*

Coupling for $(1,4)$

| $(1,4)$ | 1 | 2 | 3 | |
|---|---|---|---|---|
| 2 | $\frac{1}{3}$ | | | $\frac{1}{3}$ |
| 3 | | $\frac{1}{3}$ | | $\frac{1}{3}$ |
| 4 | | | $\frac{1}{6}$ | $\frac{1}{6}$ |
| 6 | | | $\frac{1}{6}$ | $\frac{1}{6}$ |
| | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | |

Coupling for $(3,4)$

| $(3,4)$ | 1 | 2 | 3 | |
|---|---|---|---|---|
| 2 | $\frac{1}{3}$ | $\frac{1}{6}$ | | $\frac{1}{2}$ |
| 3 | | $\frac{1}{6}$ | $\frac{1}{3}$ | $\frac{1}{2}$ |
| | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | |

Optimum coupling for $(1,4)$

| $(1,4)$ | 1 | 2 | 3 | |
|---|---|---|---|---|
| 2 | | $\frac{1}{3}$ | | $\frac{1}{3}$ |
| 3 | | | $\frac{1}{3}$ | $\frac{1}{3}$ |
| 4 | $\frac{1}{6}$ | | | $\frac{1}{6}$ |
| 6 | $\frac{1}{6}$ | | | $\frac{1}{6}$ |
| | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | |

Table 1: couplings

*Then we call Algorithm 6 to compute* $\gamma_\lambda^\Omega$*. It checks all the couplings and the coupling for* $(1,4)$ *is found not optimal for* $TP(d,\mathbf{P}(1,\cdot),\mathbf{P}(4,\cdot))$*. The optimal shipping function for the current cost function d is shown in the third table of Table 1. It is set to be the new coupling for* $(1,4)$*. Then Algorithm 6 is called again. It then checks all the couplings and found the coupling for* $(1,4)$ *is now optimal. However couplings for other pairs may not be optimal, and therefore the algorithm may continue until all the couplings are optimal.*

## 4 Concurrent algorithm

There are two phases in the algorithm. In the first phase, we create several threads to initialize the couplings as shown in Algorithm 1. We divide the pairs in the set *toCompute* into disjoint subsets and each thread is working on one subset.

**Algorithm 1:** Couplings initializations

**Data**: Markov chain $\mathcal{M} = (S, A, \mathbf{P}, l)$;
  $toCompute = \{(s,t) \in S \times S | l(s) = l(t) \wedge s \neq t\}$; work(i) $\subset$ $toCompute$ and $\cup_i work(i) = toCompute$

```
1 Thread i ( work(i) ):
2 for all (s,t) ∈ work(i) do
3 |   pick ω ∈ P(s,·) ⊗ P(t,·);
4 |   Ω[(s,t)] ← ω;
5 end
```

In the second phase of the algorithm, we parallelize the matrix computation in the method *Discrepancy*. The matrix $\mathbf{A}$ is divided into several rows, where each thread deals with several contiguous rows of $\mathbf{A}$. Each thread is executing Algorithm 2 and it waits at the barrier for all the other threads. The last thread which reaches the barrier will execute Algorithm 3. The computation result of Algorithm 3 will decide if all the threads should continue the computation or not by setting the shared boolean variable to $true$ or $false$.

If we combine the results as described in Algorithm 3, we need to create threads in every iteration in the second phase. In order to create threads just once, Algorithm 3 is modified into Algorithm 4 where we also check if the stop criterion of the second phase is satisfied. The complete concurrent algorithm is presented in the appendix.

**Algorithm 2:** Concurrent calculation of linear system

**Data**: matrix $A_{m,n}$, vectors $\boldsymbol{B}$, $\boldsymbol{X}$ and $\boldsymbol{C}$, $G \leftarrow \emptyset$, a barrier shared by all the threads and a boolean called loop which is shared by all the threads

1 $G' \leftarrow S \times S \setminus G$;
2 $A \leftarrow (\mathbf{P}(s,t))_{s,t \in G'}$ ;
3 $b \leftarrow (\sum_{t \in G} \mathbf{P}(s,t))_{s \in G'}$;
4 Thread i (startRow, endRow)
5 **while** *!loop* **do**
6 　**for** $\forall i, startRow \leq i < endRow$ **do**
7 　　$c_i = \lambda(\boldsymbol{A_i X} + b_i)$
8 　**end**
9 　barrier.wait();
10 **end**

**Algorithm 3:** Combine the results

**Data**: vectors $\boldsymbol{X}$, $\boldsymbol{C}$;a boolean called loop which is shared by all the threads;

1 loop = true;
2 **for** *int* $i = 0; i < n; i++$ **do**
3 　**if** $|c_i - x_i| > \epsilon$ **then**
4 　　loop = false;
5 　**end**
6 　$x_i = c_i$; //change $\boldsymbol{X}$ to be $\boldsymbol{C}$
7 　$c_i = 0.0$; //reset entries of $\boldsymbol{C}$ to be 0.0;
8 **end**

**Algorithm 4:** Combine the results

```
   Data: vectors X and C; a boolean called loop which is shared by all the
         threads; Markov chain M = (S, A, P, l); the set of state pairs
         toCompute;
 1 loop = true;
 2 for int i = 0; i < n; i++ do
 3   | if |c_i − x_i| > ϵ then
 4   |   | loop = false;
 5   | end
 6   | x_i = c_i; //change X to be C
 7   | c_i = 0.0; //reset entries of C to be 0.0;
 8 end
 9 if loop then
10   | while ∃(s, t) ∈ toCompute. Ω[(s, t)] not optimum do
11   |   | ω ← optimum shipping function for TP(d, P(s, ·), P(t, ·));
12   |   | Ω[(s, t)] ← ω;
13   |   | loop = false;
14   |   | break;
15   | end
16 end
```

## 5 Implementation in Java

Figure 4 illustrates the class diagram of the sequential algorithm. Since the transportation problem is an important part of the algorithm, we implement the **TPLP** class. The **TPLP** class has three fields, namely *source*, *destination* and *schedule*. *source* is an array of *double*s which stores the supplies and similarly *destination* stores the demands. *schedule* is a two dimensional array of *double*s which stores the coupling. An initial coupling will be computed and stored in *schedule* when the method *initialSolution* is called. The method *initialSolution* uses the northwest corner rule ([7]) to generate an initial schedule of the transportation problem. The method *isOptimal* checks if the current coupling is optimal for a given cost matrix. The method *getOptimal* computes the optimal schedule given a cost matrix, in which we use **SimplexSolver** of the *org.apache.commons.math*3*.optim.linear* to obtain the optimal solution.

We implement a **SimpleMatrix** class to do simple operations such as returning a row of a matrix. We also have a *Pair* class which represents a pair of states.

In the **Computation** class, the private method *getBisDist* implements Algorithm 7 and the other private method *computeDescrepancy* is the implementation of Algorithm 6. A labelled Markov chain is represented by the number of

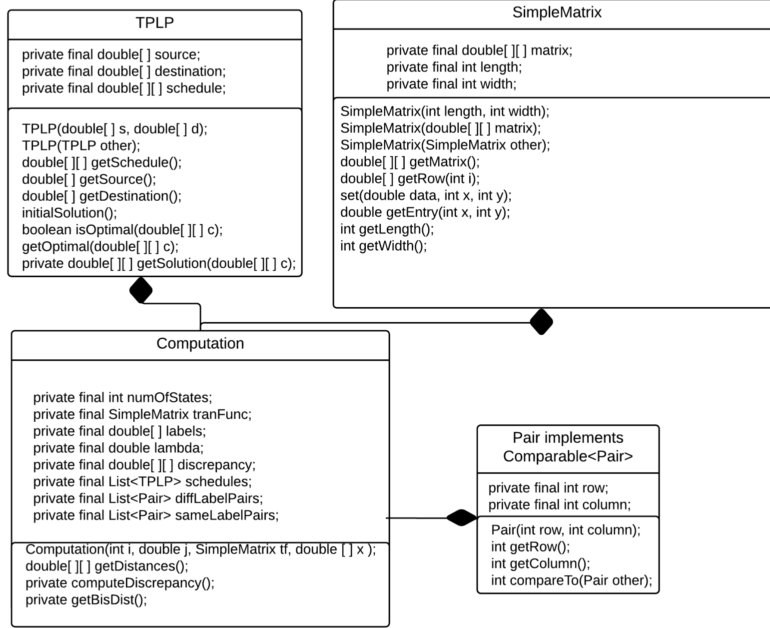



Fig. 4: Sequential Program Class Diagram.

states, an array of real numbers as the labelling function and a transition probability matrix. In order to compute the distances of a labelled Markov chain, we can construct a *Computation* object which takes the input Markov chain and the discount factor. We can call the method *getDistances* on the **Computation** object to get the matrix of bisimilarity distances.

In the implementation of the concurrent algorithm, we use a *CyclicBarrier* object and a *AtomicBoolean* object. Each thread is executing Algorithm 2 and it waits at the barrier for all the other threads coming to the barrier. The last thread which reaches the barrier will execute Algorithm 4, which decides if all the threads should continue the computation or not by setting the shared boolean variable to $true$ or $false$.

## 6 Correctness test

We generate thousands of random labelled Markov chains using the Erdos Renyi model [8] and compute the bisimilarity distances using the algorithm in [3]. We then use our sequential and the concurrent programmes to compute the distances for the same set of Markov chains. We save the results in files and use $diff$

command to compare these results. For the concurrent algorithm, we also test the computation by using different number of threads.

| number of states | 5 | 10 | 20 | 40 |
|---|---|---|---|---|
| number of Markov chains | 1000 | 200 | 50 | 50 |

The table above shows how many Markov chains we generate randomly corresponding to the number of states. The accuracy of the distance is set to be 0.000001 ($10^{-6}$) and the discount factor $\lambda$ is set to be 0.99. For the concurrent algorithm, we test with different number of threads, namely 5, 10, 20, 40.

The results generated by the sequential and the concurrent programmes are all the same. Compared to the results of reference algorithm, there are a few entries (around 4 in 1000) which have tiny differences of 0.000001 ($10^{-6}$).

## 7 Performance test

### 7.1 Test Set-up

We generate random labelled Markov chains using the Erdos Renyi model and compute the bisimilarity distances using the sequential algorithm in the concurrent algorithm presented in the previous sections. We record the time spent on each computation and compare the average time spent on each size of the Markov chain.

| number of states | 5 | 6 | 7 | 8 |
|---|---|---|---|---|
| number of Markov chains | 200 | 200 | 200 | 200 |

The table above shows that our measurement has been made on a collection of Markov chains varying from 5 to 8 states. For each $n = 5, ..., 8$, we randomly generate 200 Markov chains. The accuracy of the distance is set to be 0.000001 ($10^{-6}$) and the discount factor $\lambda$ is set to be 0.99. For the concurrent algorithm, the number of threads under the test varies from 2 to 32.

### 7.2 Results

Figure 5 shows the average running time on the Markov chains with 5 states versus the number threads. The sequential algorithm takes around 4.618 ms to compute one instance, while the concurrent algorithm no matter how many threads there are takes more than 21 ms. It is obvious that the concurrent algorithm is much slower than the sequential one.

Figure 6 shows the relationship of average overhead of creating threads and the number of threads when the number of states is in the range of 5 to 8. The

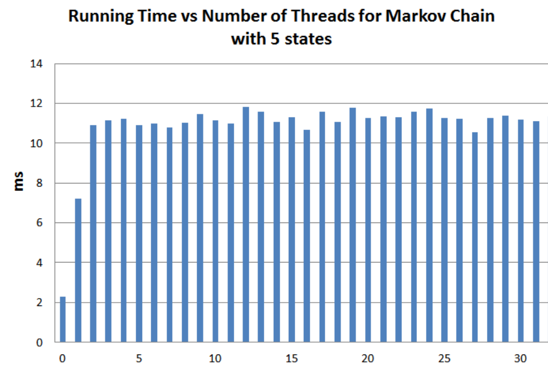



Fig. 5: The bar chart shows the running time versus the number of threads for a Markov Chain with 5 states. The vertical axis is the running time in milliseconds. The horizontal axis is the number of threads in the range of 1 to 32. The data of 0 threads corresponds to the sequential programme.

time spent on creating the threads is almost the same for different number of states when the number of threads is from 1 to 12. It can be seen from the graph that the larger the number of states is, the more overhead there is. It is due to the fact that it takes more time to divide the work among the threads when there are more states.

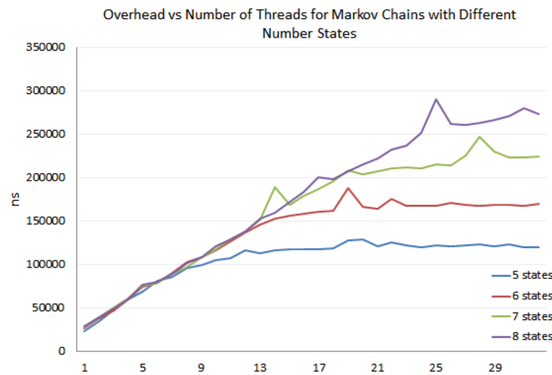



Fig. 6: The line chart shows the overhead of creating threads versus the number of threads for Markov Chains with 5-8 states. The vertical axis is the running time in nanoseconds. The horizontal axis is the number of threads in the range of 1 to 32.

Figure 7 shows the average running time versus the number threads in computing the Markov chains with 5 to 8 states. As can be seen in the graph, regardless of the number of states the sequential program always performs bet-

ter than the concurrent program. Each data point is the result of 200 runs. The standard deviation of the sequential programme is approximately 10ms for any number of states. The table below shows the standard deviation in milliseconds for the concurrent program when the number of states is from 5 to 8.

| number of states | 5 | 6 | 7 | 8 |
|---|---|---|---|---|
| overhead | 24 | 40.5 | 63 | 114.5 |

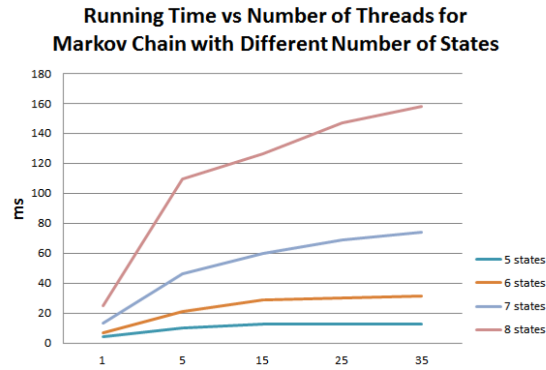



Fig. 7: The line chart shows the running time in millisecond versus the number of threads on the Markov chains with 5-8 states. Each line represents the running time on Markov chains with a fixed number of states.

The results of the concurrent program are poor compared with the sequential program. To see whether the concurrent initialization provides any performance gain, we test the concurrent implementation by replacing the concurrent initialization with the sequential one. It turns out that concurrent initialization does not actually improve the performance. Figure 8 compares the running time on the Markov chain with 5 states of the concurrent program and the one with a sequential initialization. It shows that the concurrent initialization slows down the program. It is probably because that there is a lot more time spent on creating new threads and bookkeeping than the time saved by the concurrent execution.

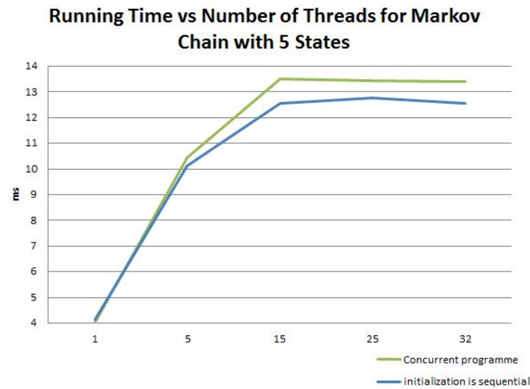



Fig. 8: The line chart compares the performance of the concurrent program and the one with sequential initialization on the Markov chain with 5 states. The green line is the program with sequential initialization while the blue one is the one which has concurrent initialization.

### 7.3 Analysis

The results indicate that the concurrent program is 4 to 5 times slower than the sequential program. We are keen to learn why the concurrent matrix computation does not gain us any performance. The possible answer might be that the sizes of the matrices in our test are so small that the time spent on bookkeeping and threads scheduling exceeds the speed-up brought by concurrent computation. To verify it, we run a test on the matrix computations to see the relationship of the running time versus the size of the matrices.

We take out the part of program which does the matrix computation from both the sequential program and the concurrent program. We then generate matrices of different sizes and test the performance of them separately. Figure 9 illustrates the running time of computation versus the number of threads for different sizes of matrices. It is noticed in the graph that the concurrent computation is slower than the sequential one when the size of the matrix is less than 128. The performance of the concurrent algorithm turns out to be better than the sequential one as the size of the matrix grows.

The above test well explains why our concurrent algorithm is much slower than the sequential algorithm. In our performance test, the Markov chains under the test have 5 to 8 states which fixes the matrix size to be at most 64. However the above test shows that the concurrent algorithm is slower than the sequential one when the matrix size is less than or equal to 128. In addition, in the concurrent algorithm which computes the bisimilarity distances there might be extra overhead by taking into account the use of the barrier.

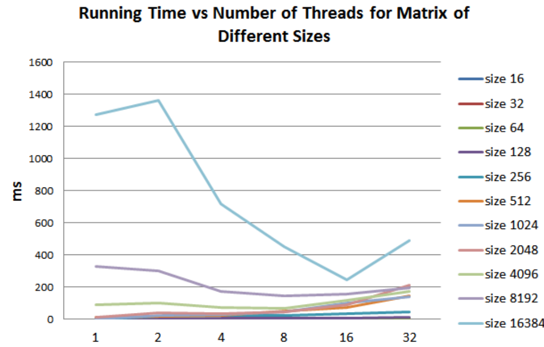



Fig. 9: The line chart shows the running time in milliseconds versus the number of threads for different matrix sizes. The matrix doubles in size from 16 to 16384.

We increase the number of states to 50 and find the concurrent program is still slower than the sequential program. The sequential program takes about 23 minutes on average (10 Markov chains under the test) and the concurrent program with 64 threads takes about 62 minutes on average.

## 8 Conclusion

Bisimulation distance is a pseudo-metric which could be used to describe the similarity of two probabilistic processes. Bacci et al. claimed that the on-the-fly algorithm they came up with was more efficient than other algorithms ([3]). In this paper, we study if it is possible to improve the performance of the algorithm by making it concurrent.

Our concurrent implementation has a very limited amount of concurrency which is due to the data dependencies in the second phase of the algorithm. It is critical to determine if the algorithm can be further parallelized. We tested the correctness of both the sequential and concurrent implementations. We are confident that the implementations are correct since the results are almost the same compared with the results of the algorithm by van Breugel and Worrell.

We tested the performance of the concurrent implementation and the sequential one on a machine in the Intel MTL which has 80 cores. The test shows that there is no performance improvement in our concurrent implementation. We then ran a test on the computation of the matrix and verified that the concurrent program would be slower when the matrix size is less than 128. We changed the test settings and found the concurrent implementation did not outperform the sequential one when the Markov chains have 50 states. The test results seem to suggest that the algorithm in [3] is not easy to be parallelized.

**Acknowledgement**

We thank the team at Intel's Multicore Testing Lab for providing us access to their machine.

## References

1. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. In: Conference Record of the 16th ACM Symposium on Principles of Programming Languages (ACM). (1989) 344–352
2. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theoretical Computer Science **318**(3) (2004) 323 – 354
3. Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: On-the-fly exact computation of bisimilarity distances. In Piterman, N., Smolka, S.A., eds.: Proceedings of 19th Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 7795 of Lecture Notes in Computer Science. Springer (2013) 1–15
4. van Breugel, F., Worrell, J.: Approximating and computing behavioural distances in probabilistic transition systems. Theoretical Computer Science **360**(1–3) (2006) 373 – 385
5. Hillier, F.S., Lieberman, G.J.: Introduction to Operations Research, 4th Ed. McGraw-Hill Higher Education, Boston , USA (2005)
6. Chen, D., van Breugel, F., Worrell, J.: On the complexity of computing probabilistic bisimilarity. In Birkedal, L., ed.: Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS). Volume 7213 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 437–451
7. Rao, S.S.: Engineering Optimization: Theory and Practice. Wiley; 4 edition (2009)
8. Erdős, P., Rényi, A.: On the strength of connectedness of a random graph. Acta Mathematica Hungarica **12**(1) (1961) 261–267

## A Sequential algorithm

**Algorithm 5:** Computation of Bisimilarity Distances

**Data**: Markov chain $\mathcal{M} = (S, A, \mathbf{P}, l)$; discount factor $\lambda$

1. $\Omega \leftarrow \emptyset$; $d \leftarrow$ *empty*; $G \leftarrow \emptyset$;
2. **for** *all* $(s,t) \in S \times S$ **do**
3.     **if** $l(s) \neq l(t)$ **then**
4.         $d(s,t) \leftarrow 1$;
5.         $G \leftarrow G \cup (s,t)$;
6.     **end**
7.     **else if** $s = t$ **then**
8.         $d(s,t) \leftarrow 0$;
9.     **end**
10.     **else**
11.         pick $\omega \in \mathbf{P}(s,\cdot) \otimes \mathbf{P}(t,\cdot)$;
12.         $\Omega[(s,t)] \leftarrow \omega$;
13.     **end**
14. **end**
15. Discrepancy($\lambda$);
16. **while** $\exists (s,t) \in S \times S.\ \Omega[(s,t)]$ *not optimum* **do**
17.     $\omega \leftarrow$ optimum shipping function for $TP(d, \mathbf{P}(s,\cdot), \mathbf{P}(t,\cdot))$;
18.     $\Omega[(s,t)] \leftarrow \omega$;
19.     Discrepancy($\lambda$);
20. **end**

**Algorithm 6:** Discrepancy

**Data**: discount factor $\lambda$

1. $G' \leftarrow S \times S \setminus G$ ;
2. $\mathbf{A} \leftarrow \Omega[(s,t)](s',t')_{(s,t),(s',t') \in G'}$ ;
3. $\mathbf{b} \leftarrow (\sum_{(s',t') \in G} \Omega[(s,t)](s',t'))_{(s,t) \in G'}$;
4. $\bar{\mathbf{x}} \leftarrow \mathbf{x} = \lambda \mathbf{A}\mathbf{x} + \lambda \mathbf{b}$;
5. **for** $(s,t) \in G'$ **do**
6.     $d(s,t) \leftarrow \bar{\mathbf{x}}(s,t)$;
7. **end**

## B Concurrent algorithm

**Algorithm 7:** Computation of Bisimilarity Distances

```
   Data: Markov chain M = (S, A, P, l); discount factor λ; number of threads
         numOfThreads
 1 Ω ← ∅; d ← empty; G ← ∅; toCompute ← ∅;
 2 for all (s, t) ∈ S × S do
 3     if l(s) ≠ l(t) then
 4         d(s, t) ← 1;
 5         G ← G ∪ (s, t);
 6     end
 7     else if s = t then
 8         d(s, t) ← 0;
 9     end
10     else
11         toCompute ← toCompute ∪ (s, t);
12     end
13 end
14 divide toCompute into subsets, each one is work(i);
15 create numOfThreads threads;
16 Couplings initializations (Algorithm 1);
17 create numOfThreads threads;
18 Concurrent calculation of linear system (Algorithm 2);
19 Combine the results (Algorithm 4);
```