# The power of two paths in grid computing networks

Wouter ibens
University of Antwerp
Supervisor: Prof. dr. Benny Van Houdt

May 28, 2012

# Abstract

In ring structured distributed systems, busy nodes will forward new jobs to other nodes. This thesis focusses on the algorithms for choosing a successor node for a job. ...

## Acknowledgements

This thesis is not only the work of myself, I could never accomplish this without the people around me. I would like to take this opportunity to thank some of them specifically.

First and foremost, my supervisor Professor dr. Benny Van Houdt. Firstly, he taught me a whole new field in the area of computer science, the part that interested me the most during the course of my studies. Secondly, for being my supervisor: he helped me out when he could and suggested ideas when I was stuck. I was always welcome to drop by and reflect thoughts.

I would also like to thank my family, especially my parents. They gave me the chance to complete my education without worrying just one second about the financial cost. They gave me the freedom of making my own choices and motivated me when I needed it. My sister Anneleen should be mentioned for proofreading this thesis and other tasks in english during my education.

Also, my friends have been a great help to me. Playing games, eating toghether every evening and just having fun together is what made the past 5 years with no doubt the best years of my life so far. Thank you all.

Finally, I would like to thank my girlfriend Nicky. Although she might have caused some failed exams at the beginning of our relationship, she has been a great help further on. She motivated me to take hard but rewarding options when easy ones were available. A girl that understands me and never fails to cheer me up.

# Contents

# Introduction

This thesis researches the behaviour of forwarding algorithms in a ring-structured distributed system. Section 1 precisely describes the setup of the system. It specifies the general assumptions made in this document and gives a short overview of the different algorithms we have reviewed.

Section 2 gives a short introduction about the simulator we wrote and how to use it. Furthermore, it contains the results of the tests, performed by the simulator.

In the next part, section 3, we reviewed the output of the simulator. Using numerical algorithms we try to match our results with a Markov Chain model.

Finally, section 4 contains the conclusions and other thoughts on the algorithms.

** COMPLETE LATER ON

# 1 Setup

We are using a ring-structured distributed system of $N$ nodes. Each node is connected to two neighbours, left and right. The purpose of these nodes is to process incoming jobs. When a node is busy while a job arrives, it must forward the incoming job to another node. When a job has visited all nodes and none of them was found idle, the job is dropped.

Jobs have an arrival time, a length, the ID of the first node and optional metadata. They arrive at each node indepentently as a poisson process at rate $\lambda$. Their length is exponentially distributed with mean $\mu$ (unless otherwise noted, assume $\mu = 1$). Although each job has a length, this length may not known in advance. Finally, the metadata is optional and may be used by the nodes to pass information among the job (e.g. a list of visited nodes).

Nodes can use different algorithms to determine whereto a job will be forwarded. The performance of these algorithms is the main focus of this thesis. Different techniques will be discussed and simulated. Afterwards, some results of the simulation will be validated using numerical techniques. Note that the cost of forwarding a job is neglected. Together with the presumption a job must visit each node before being dropped, this means a job arriving at any node will be processed if and only if at least one server is idle.

The performance of a forwarding algorithms is measured by the average number of nodes a job must visit before being executed. The goal of the algorithms is to minimize this number by spreading the load evenly along the ring.

## 1.1 Forwarding algorithms

Nodes that must forward a job will choose another node of the ring. Nodes have no information about other nodes, so it has no idea whitch nodes are idle or busy. The algorithms are grouped into two categories: forward to neighbour and forward anywhere. The first techniques allows a busy node to forward an incoming job to either its left or its right neighbor, where the latter may forward these jobs to any node in the ring. Since the amount of dropped jobs is equal for each forwarding algorithm. These jobs will be ignored when computing the

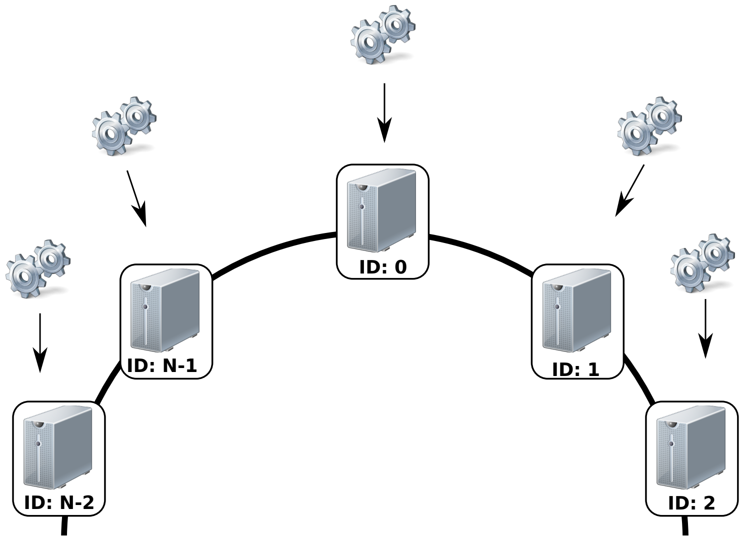


Figure 1: A ring structured network

average number of forwards. One should note that the loss rate of jobs in the system is the same as the Erlang-b loss rate.

## 1.2 Forward to neighbour

**Forward right**

A busy node using this technique will forward a job to its right neighbour. The job will keep travelling in this direction until an idle node is found, where it will be processed or it has visited all nodes, when it will be dropped. Unless otherwise noted, this algorithms is used as baseline in all further tests. This algorithm does not use any space in the job's metadata, nor must nodes save states.

**Left/Right forward**

A variant to the previous algorithm is the Left/Right forward technique. Instead of forwarding each job to its right neighbour, a busy node will alternate the direction after forwarding such a job. To avoid a job coming back, this initial direction is saved in the job's metadata. Busy nodes receiving a job from another neighbour must forward it the same direction as specified in the job's metadata. This algorithms requires the 1 bit representing the direction in the job's metadata, it also needs the save 1 bit state information in each nodes to keep track of the last direction a new job was forwarded to.

**Random Left/Right forward with parameter $p$**

This technique is a variant of the Left/Right forward algorithm. However, instead of alternating the direction for each new job, a node will forward a job to its right with probability $p$ and to its left with probability $1 - p$. As the previous technique, the direction is saved in the job's metadata and subsequent nodes must maintain this direction when forwarding. On the contrary, nodes do not need to save state information.

#### Position-dependant forwarding

As shown in figure 1, each node in the ring has an unique ID. Except the for node with id 0 and $N-1$, neigbouring are succeeding. When nodes uses this algorithm, nodes will always forward a new job in the same direction: to the right when the node's id is even, to the left otherwise. As previous algorithms, the direction is saved in the job's metadata and this direction must be used if other nodes must forward the job. The initial direction of incoming jobs can be derived from the node's ID, therefore the node needs no state information.

### 1.3 Forward anywhere

The ring strucure can be used in real networks, however in many cases the ring is no more than a virtual overlay over another structure (e.g. the internet). In these networks each node is able to connect each other node directly and more sophisticated forwarding algorithms can be used.

#### Random unvisited

The Random unvisited algorithm is the most basic algorithm in this category. Everytime a job must be forwarded, a list of unvisited nodes is generated and a random node is choosen from that list. The current node is added to the list of visited nodes, which is found in the job's metadata. This list can be saved using $N$ bits.

#### Coprime offset

Another algorithm is Coprime offset. This algorithm generates a list of all numbers smaller than $N$, and coprime to $N$. The first time a job is forwarded the next number of this list is selected. This is the job's forward offset and saved in the its metadata. When a job is forwarded, it is sent exactly this many hops farther. Because this number and $N$ are coprime, it will visit all nodes exactly once before being returned to its originating node.

Example: Consider a ring size of $N = 10$ in which every node is busy. The list of coprimes is than generated: $1, 3, 7, 9$. Assume a job arrives at node 3 and the last time node 3 forwarded a job it was given offet 1. Because this node is busy, the next number on the list (3) is selected and saved in the job's metadata. All nodes are busy so the job visits these nodes before being dropped: 3 (arrival), $6, 9, 2, 5, 8, 1, 4, 7, 0$. Node 0 will drop the job because the next node would be 3, which is the node on wich the job arrived.

Because the list of coprimes can be regenerated each time a job arrives, it must not be saved. Both the job and the node must keep an index to a coprime. The size of this list $\tau(N)$ thus an index to an element in this list requires $\lceil log_2 \tau(N) \rceil$ bits.

### Random Coprime offset

The Random Coprime offset algorithm is almost equal to Coprime offset. The difference between them is the decision of the offset value. Where it is the next number on the list in Coprime offset, a random value is taken from the list when using Random Coprime offset. As Coprime offset, the offset must be saved in the job's metadata. Since the offset for incoming jobs is choosen at random, the nodes do not need to keep state information.

## 1.4 Properties of Algorithms

We have reviewed some of the basic properties for each algorithm, they can be found in table 1. The function $\tau(N)$ represents the number of values smaller than $N$ and coprime to $N$.

$$\tau(N) = \sum_{0<p<N, p \perp N} 1$$

Note that this value is upper bound by $N-1$ and equals $N-1$ when $N$ is prime.

| | Size known | First forward | Possible paths | Space requirement (job) | Space requirement (node) |
|---|---|---|---|---|---|
| Forward Right | N | 1 | 1 | 0 | 0 |
| Left/Right forward | N | 2 | 2 | 1 | 1 |
| Random Left/Right forward (p) | N | 2 | 2 [1] | 1 | 0 |
| Position dependant forward | N | 1 | 1 | 1 | 0 |
| Random unvisited | Y | $N-1$ | $(N-1)!$ | $N$ | 0 |
| Coprime offset | Y | $\tau(N)$ | $\tau(N)$ | $\lceil log_2 \tau(N) \rceil$ | $\lceil log_2 \tau(N) \rceil$ |
| Random Coprime offset | Y | $\tau(N)$ | $\tau(N)$ | $\lceil log_2 \tau(N) \rceil$ | 0 |

Table 1: Properties of forwarding algorithms

**Size known** Represents whether the size of the ring must be known to the nodes in order to function.

**First forward** When the node on which the incoming job arrives is busy, this number represents the number of candidates to which the node can forward the job.

**Possible paths** Assuming the system is saturated, this number represents the number of possible paths a job can travel until being dropped. The probabilty of each path is the same (unless otherwise noted).

**Space requirement (job)** This value represents the number of bits required in the job's metadata.

**Space requirement (node)** This value represents the number of bits required to save the node's state information.

[1] For Random Left/Right forward for which $p \neq 0.5$, the probabilty of each path is different.

# 2 Simulation

To evaluate the different algorithms discussed in the previous section, 2 methods will be used. Firstly using a simulation, the second method is the numerical evaluation of this simulation using MATLAB. The validation method is further discussed in section 3.

The simulation is accomplished using a custom simulator. A continuous time simulator is written in C++, using no external requirements but the STL and OpenMP [2]. The source code of the simulator can be found in appenix A or at `http://code.google.com/p/powerofpaths/`.

The simulator can be controlled using a command line interface of which the usage is described below:

```
Usage: -r -s long -j double -a double -n long -p long -l long -t
    long -h type
        -r      Random seed
        -s      Set seed                        (default: 0)
        -j      Job length                      (default: 1.0)
        -a      Load                            (default: 1.0)
        -n      Ring size                       (default: 100)
        -c      Processing units per node       (default: 1)
        -p      Print progress interval         (default: -1 -
            disabled)
        -l      Simulation length               (default: 3600)
        -t      Repetition                      (default: 1)
        -h      Print this help
         type   right | switch | randswitch | evenswitch | prime |
             randprime | randunvisited | totop
```

Listing 1: Simulator usage description

## 2.1 Measure

The goal of the algorithms is to distribute the jobs evenly along the ring. This implies that the number of nodes a job must visit should be low. As a measure for our experiments, we will be using the average number of times a job is forwarded before it is executed. Since the number of forwards of a job that could not be executed is the same for each algorithm (i.e. $N-1$: traversing each link but one), and the loss rate of each algorithm is the same (because forwards are instantaneous), we will not take these jobs into account when computing the average.

It is clear that when the system load approaches 0, the probability that a node is busy will also approach 0 and the average number of forwards will therefore also approach 0. On the other hand, when the load approaches $\infty$, each node's probability of being busy will approach 1 and therefore the number of forwards will be $N-1$ and the job will fail. A system with load $> 1$ is called an overloaded system. For the purpose of this thesis, only loads up to 1 are discussed.

We will compare each algorithm to a baseline result. The baseline used in this thesis is the Forward right algorithm. This means that each graph will show its result relative to the Forward right results. The results given by the simulator were obtained using a ring size of 100 and using a random seed for each run.

The absolute performance of the baseline algorithm is shown in figure 2.

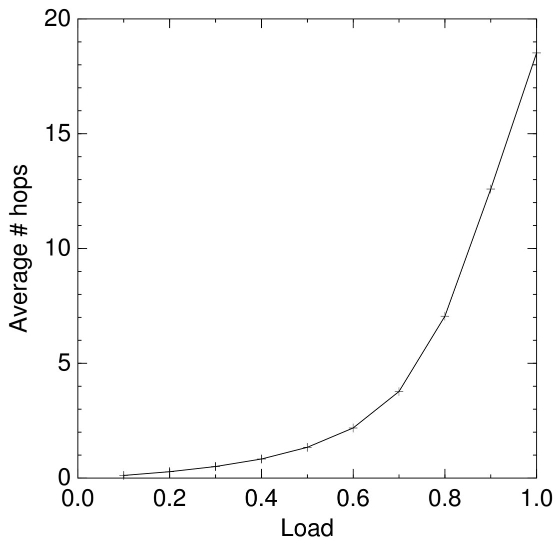



Figure 2: The Forward Right baseline result

## 2.2 Results

### Left/Right Forward

It is intuitively clear that alternating the forwarding direction of incoming jobs should distribute the load better than keeping the same direction, certainly under temporary local heavy load. Figure 3 shows the improvement made by the Left/Right forward algorithm over the Forward right method. The performance gain is at least 1% and up to over 4% under medium load.

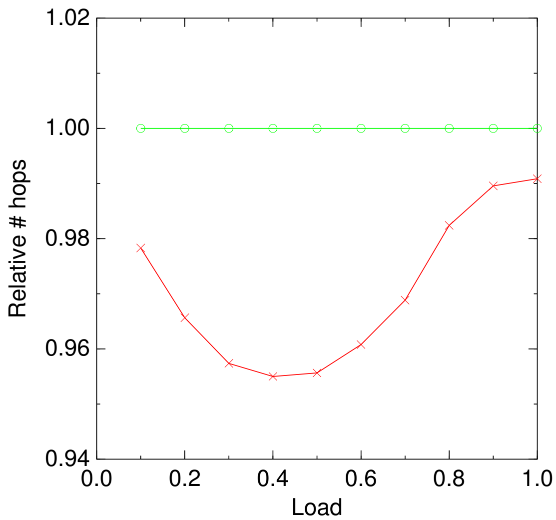



Figure 3: Left/Right

### Random Left/Right forward with parameter $p$

For $p = 0.5$, one would expect the results of this algorithm being similar to those obtained in the previous simulation. However, it seems the small change in the algorithm worsened the results significantly.

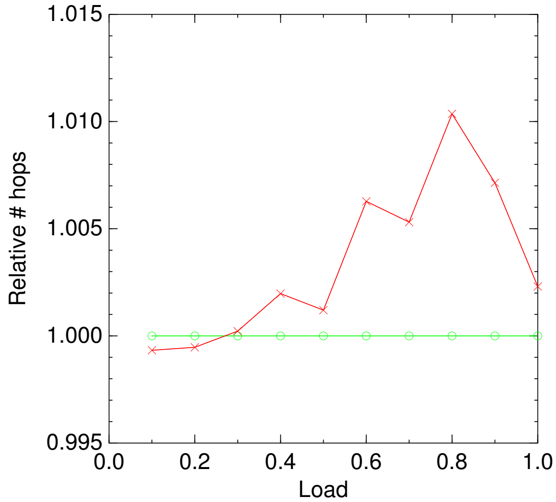



Figure 4: Random Left/Right forward with parameter 0.5

Figure 4 shows the results of this algorithm for $p = 0.5$. For $p = 0$, the algorithm is equal to the Forward Right algorithm. Hence, we should investigate how different values of $p$ influence the final results. Figure 5 shows the results obtained for $0 \leq p \leq 0.5$. We see there is no value of $p$ for which this algorithms performs better than Forward Right.

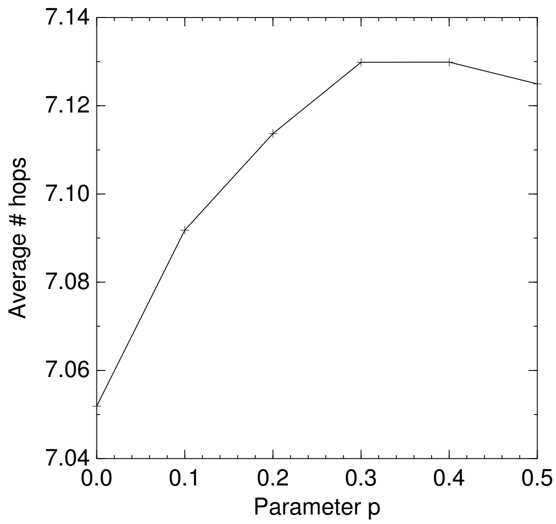



Figure 5: Random Left/Right forward with load 0.8

#### Position-dependant forwarding

This technique groups every 2 nodes into small virtual clusters. When a job arrives at a busy node, the job will be forwarded to the other node in this cluster. Jobs leaving a cluster will seem to do this in a random direction ($p = 0.5$). Since the load is concentrated per cluster instead of being distributed over the whole system, this technique performs worse than other techniques The results are represented in figure 6.

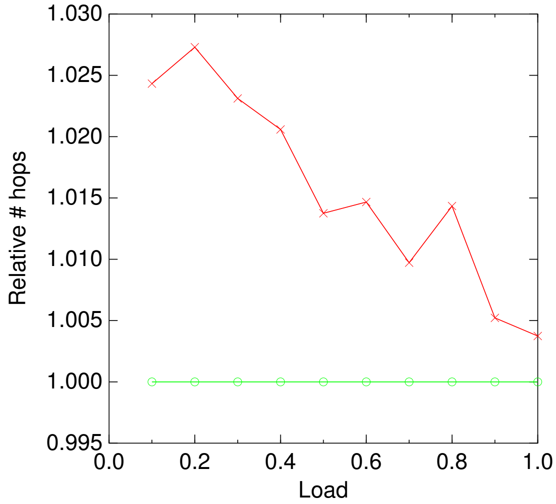



Figure 6: Position-dependant forwarding

#### Random unvisited

This algorithm is not bound to its neighbors when forwarding a job. Lifting this constraint allows a serious performance boost. ***********

This algorithm in the most straight forward and is the best performing from any of these techniques. However, it should be noted that each visited node must be stored into the job's metadata, at least $N$ bits should be available to store this information.

Figure 9 shows the results of this algorithms. It is the first algorithm tested that could forward jobs to nodes other than its neighbors. We see that lifting that constraint allows a serious performance boost.

#### (Random) Coprime offset

Two other algorithms that are not restricuted to forwarding to a neighbor are Coprime offset and Random Coprime offset. Figure 8 shows the results of both these algorithms. The difference of these algorithms with themselves and the random unvisited algorithms is not clearly visible when comparing both to the Forward Right algorithm. To put these results in perspective, we included figure **??** where Coprime offset and Random Coprime offset are depicted relative to Random unvisited. Although the difference is small, it seems the Random unvisited algorithm shows a better performance than the other two. However,

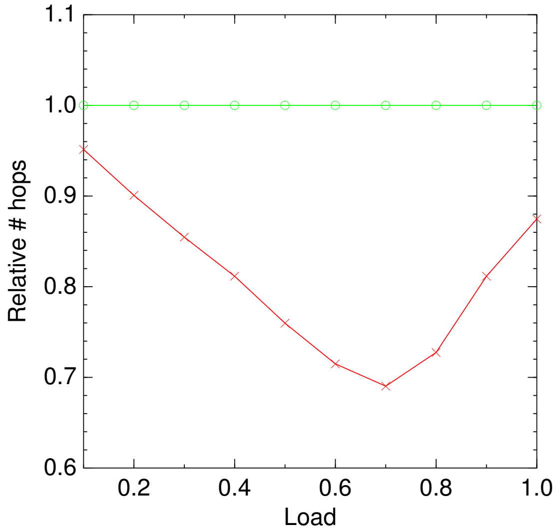
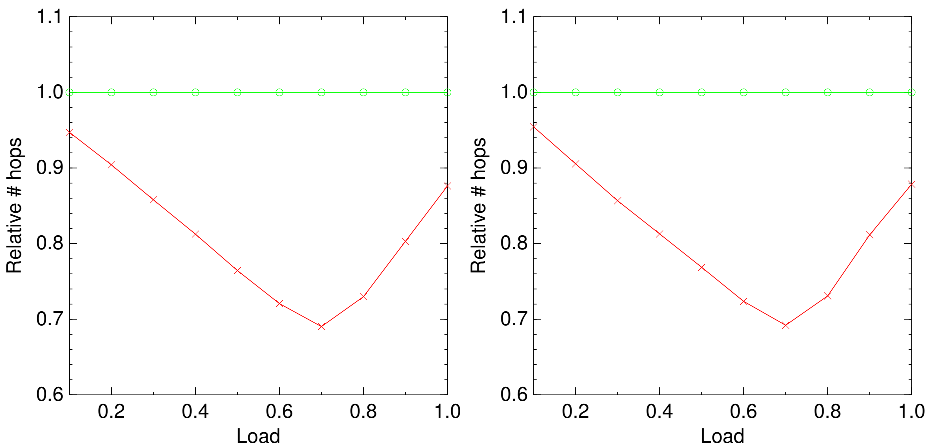



Figure 7: Random unvisited forwarding

Coprime offset and Random Coprime offset require $\lceil log_2 N \rceil$ bits in the job's metadata instead of $N$ for Random unvisited.




(a) Coprime offset

(b) Random Coprime offset

Figure 8: (Random) Coprime offset

## 2.3 Multiple execution units

So far, all simulations were executed using 1 cpu per server. This is not a realistic assumption for most distributed systems. This section is intended to research the behaviour of the algorithms when using multiple cpu's, and comparing the results to a system with 1 cpu per server, and to each other.

Per algorithm, the two simulations are performed and compared, see table 3 for more information about the tests.

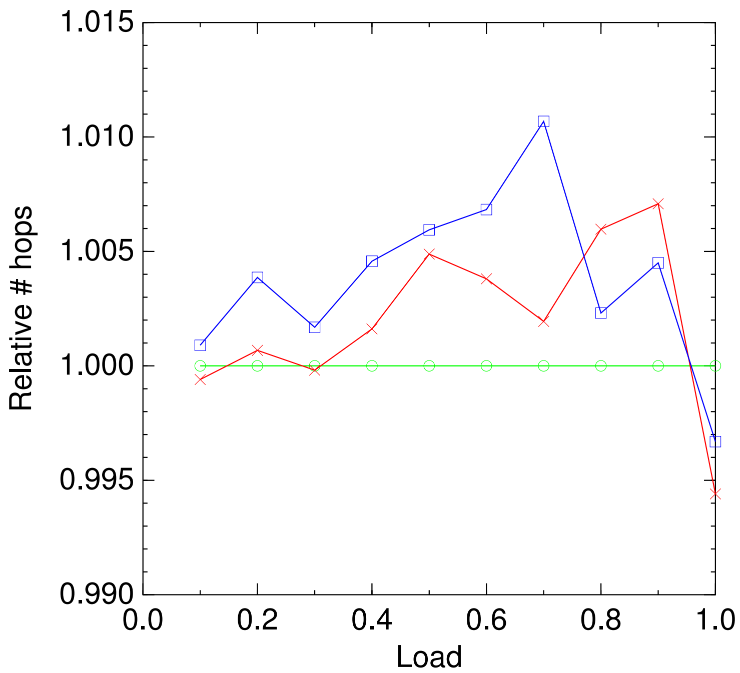



Figure 9: Random unvisited (green), Coprime offset (blue), Random Coprime offset (Red)

| | | |
|---|---|---|
| Algorithms | Forward right, Left/Right, Random left/right (0.5), Random unvisited, Random Coprime offset | |
| Ring size | 100 | 25 |
| Cpu's per node | 1 | 4 |
| Load | 0.1 – 1.0 | |

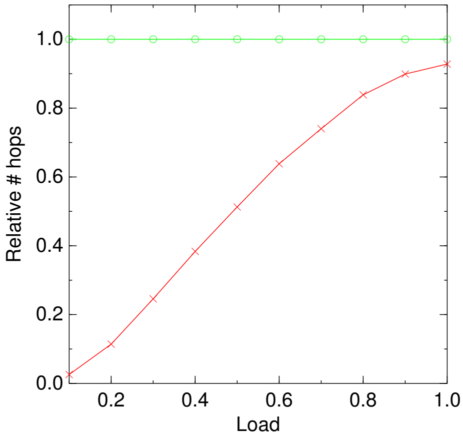
Table 2: Comparison of algorithms using multiple cpu's per server

Since the ring size for the second test is 25 instead of 100, the number of forwards of the second test will be multiplied by 4 to get meaningful results. This actually makes sense because a job that is forwarded once has encountered 4 busy cpu's. The baseline for the tests is the basic result of the algorithm found in section 2.2, where the second test is drawn relative to the basic result.

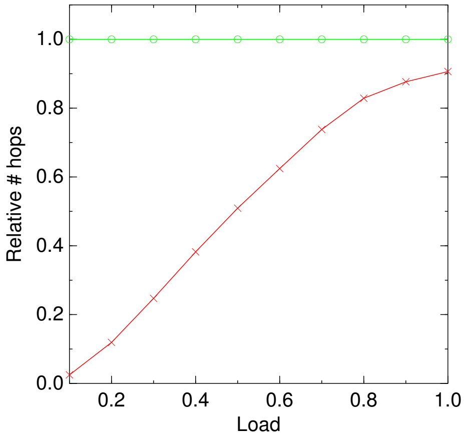
Figure 10 shows the performance of 4 cpu's versus 1 cpu for different algorithms. For each of the tested algorithms, we see the same result. This means the results of each algorithm is influenced in the same way when using multiple cpu's per server. We can use this knowledge to generalize the results of section 2.2.

Since we assume each algorithm is infuenced the same way, let us investigate one of them in depth. We will try to transform the results of an algorithm with ring size $N$ and 1 cpu per server into the results of the same algorithm with ring size $N/c$ and $c$ cpu's per server. The load of the system should be the same.

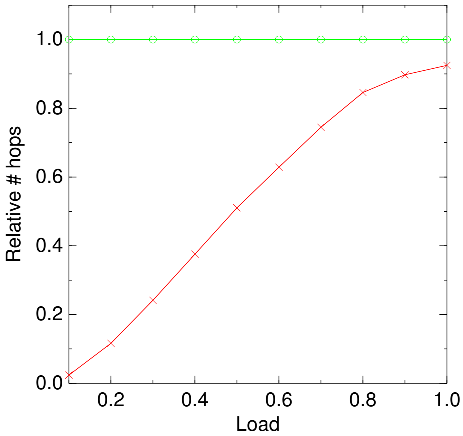
Let $d$ be the distribution of the number of forwards, stored as a rowvector. We will build a transformation vector $M$ in the form $[\lfloor 0/c \rfloor, \lfloor 1/c \rfloor, \ldots, \lfloor (N -$



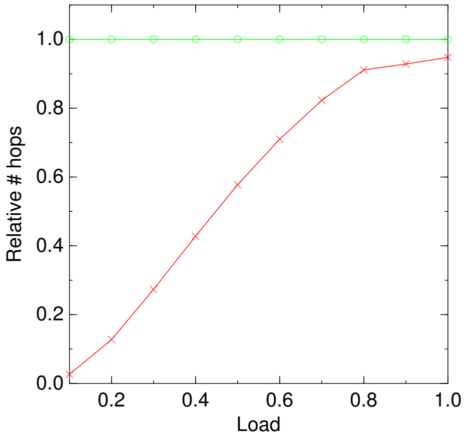

(a) Forward right



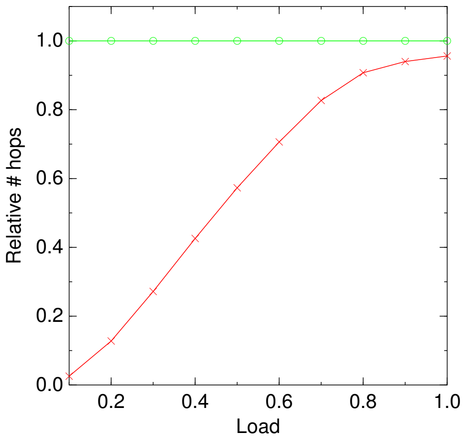

(b) Forward Left/Right




(c) Random Left/Right (0.5)




(d) Random unvisited




(e) Random Coprime offset

Figure 10: 4 cpu's per server versus 1

| # hops | Distribution | M | Result |
|---|---|---|---|
| 0 | 0.5092 | 0 | 0 |
| 1 | 0.2118 | 0 | 0 |
| 2 | 0.1082 | 1 | 0.1082 |
| 3 | 0.0609 | 1 | 0.0609 |
| 4 | 0.0363 | 2 | 0.0726 |
| 5 | 0.0224 | 2 | 0.0449 |
| 6 | 0.0142 | 3 | 0.0426 |
| 7 | 0.0091 | 3 | 0.0272 |
| 8 | 0.0058 | 4 | 0.0233 |
| 9 | 0.0037 | 4 | 0.0147 |
| Total | 0.9816 | | 0.3944 |
| Weighted total | 0.3944/0.9816 = 0.4018 | | |
| Simulation result | 0.4171 | | |

Table 3: Comparison when load=0.5

$1)/c\rfloor]'$. This vector represents the number of times a job would be forwarded if there would be $c$ cpu's per server. $d * M$ equals the average number of forwards when using such a system. To negate the dropped jobs, the results must be weighted for only the completed jobs. In our example, we will transform the results of the Forward Right algorithm using ring size $N = 10$ and 1 cpu into the results of a ring with size $N = 5$ and $c = 2$ cpu's per server, see table 3 for a worked out example and figure 11 for a comparison of the transformation and a real simulation.

# 3 Numerical Validation

** BEPAALDE ALGORITMES NIET BESPROKEN, WAAROM?

To validate the results obtained in the previous section, we modelled the scheduling-techniques into Markov Chains. Using the steady state distribution of these chains, we can derive the average number of hops and the average loss. For $N$ nodes in a ring, the markov chain consists of $2^N$ states, where the $n$-th bit represents whether the $n$-th server is busy (1) or idle (0). To optimize the computation time and memory requirements, we used sparse matrixes for the validation. The validation code is written in MATLAB, it can be found in appendix B or on `http://code.google.com/p/powerofpaths/`.

The validation of the results happens in a different environment than the simulation. Because of the non-polynomial execution time of the algorithm, the size of the ring is reduced to 10. Therefore, the results of this validation are smaller but the relative results are still relevant.

## 3.1 Forward Right

Modelling a technique into a Markov Chain is an easy operation for most algorithms. The example given below is for a ring of 3 nodes. For convenience, the states are represented by their binary form.

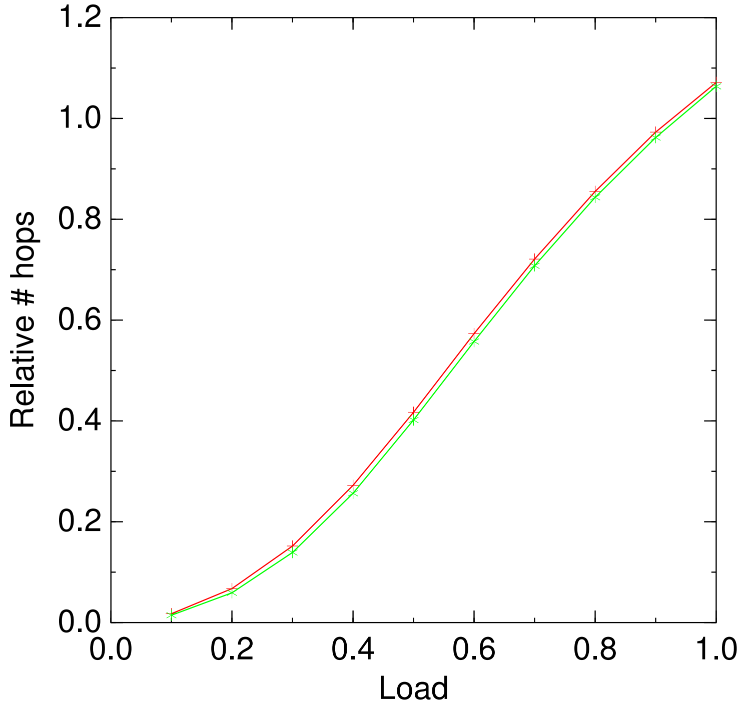



Figure 11: Multiple cpu's result derived of the result for 1 cpu (green) versus the actual simulation result (red)

$$Q = \begin{array}{c} \\ 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{array} \begin{array}{c} \begin{array}{cccccccc} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array} \\ \begin{bmatrix} -3\lambda & \lambda & \lambda & 0 & \lambda & 0 & 0 & 0 \\ \mu & -3\lambda-\mu & 0 & \lambda & 0 & 2\lambda & 0 & 0 \\ \mu & 0 & -3\lambda-\mu & 2\lambda & 0 & 0 & \lambda & 0 \\ 0 & \mu & \mu & -3\lambda-2\mu & 0 & 0 & 0 & 3\lambda \\ \mu & 0 & 0 & 0 & -3\lambda-\mu & \lambda & 2\lambda & 0 \\ 0 & \mu & 0 & 0 & \mu & -3\lambda-2\mu & 0 & 3\lambda \\ 0 & 0 & \mu & 0 & \mu & 0 & -3\lambda-2\mu & 3\lambda \\ 0 & 0 & 0 & \mu & 0 & \mu & \mu & -3\mu \end{bmatrix} \end{array}$$

Analogue to the simulation section, this method will be the baseline result in our other results.

## 3.2 Random Left/Right forward with parameter $p$

This matrix is very similar to the one above. But we need to take into account the parameters $p$ and $1-p$ instead of 1 and 0.

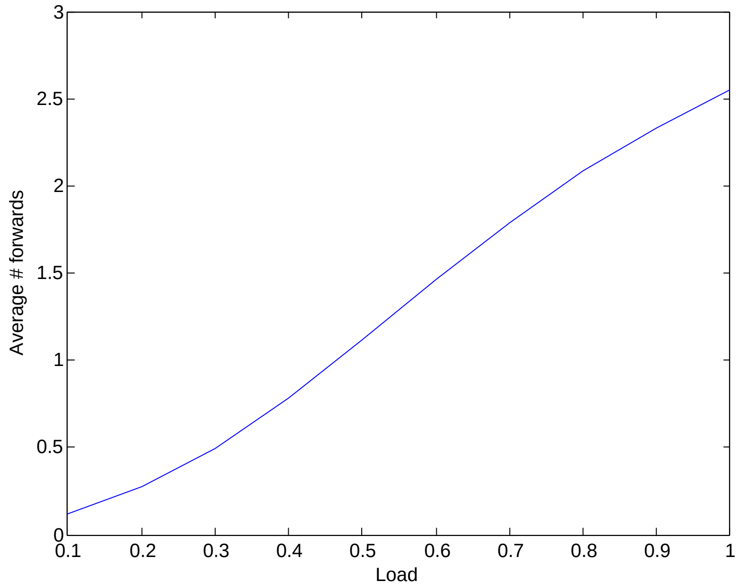



Figure 12: Validation of Forward right

$$Q = \begin{array}{c c} & \begin{array}{cccccccc} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array} \\ \begin{array}{c} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{array} & \begin{bmatrix} -3\lambda & \lambda & \lambda & 0 & \lambda & 0 & 0 & 0 \\ \mu & -3\lambda-\mu & 0 & (2-p)\lambda & 0 & (1+p)\lambda & 0 & 0 \\ \mu & 0 & -3\lambda-\mu & (1+p)\lambda & 0 & 0 & (2-p)\lambda & 0 \\ 0 & \mu & \mu & -3\lambda-2\mu & 0 & 0 & 0 & 3\lambda \\ \mu & 0 & 0 & 0 & -3\lambda-\mu & (2-p)\lambda & (1+p)\lambda & 0 \\ 0 & \mu & 0 & 0 & \mu & -3\lambda-2\mu & 0 & 3\lambda \\ 0 & 0 & \mu & 0 & \mu & 0 & -3\lambda-2\mu & 3\lambda \\ 0 & 0 & 0 & \mu & 0 & \mu & \mu & -3\mu \end{bmatrix} \end{array}$$

The lumped matrix (section 3.5) of $Q$ is equal to the lumped matrix of the example above, i.e. the matrix defines the exact same behaviour. However, for $N > 6$ the matrices and so the results of the steady state distribution begin to differ.

As in the simulation section, we have validated the results for different values for $p$. These results are shown in figure 14.

## 3.3 Random Coprime offset

Modelling this technique yields different results for various ring sizes. The performance of this algorithm is very dependant on the number of coprimes that can be used. This technique yields the same results as Forward Right for ring sizes of up 4. For $N = 3$, the matrix $Q$ is identical to Random Left/Right forward with parameter 0.5, as the coprimes of 3 are 1 and 2. Which means forwarding a job left or right, both with the same probability.

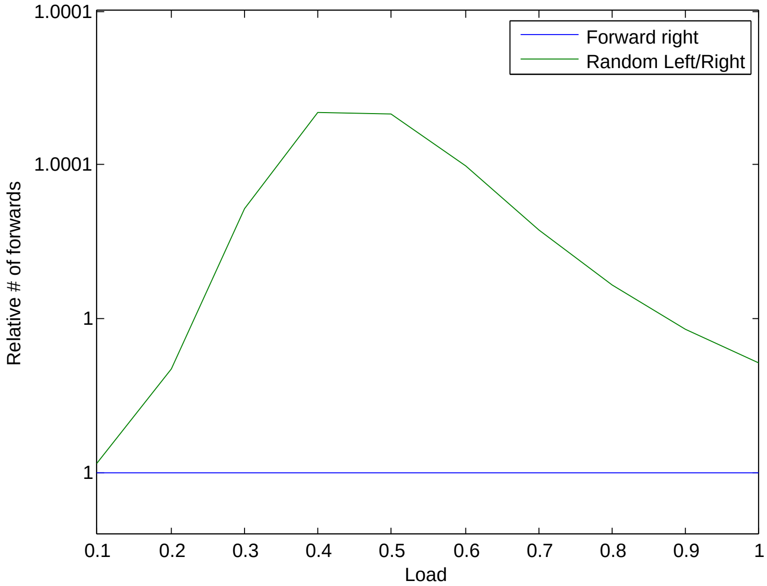



Figure 13: Validation of Random Left/Right with p=0.5

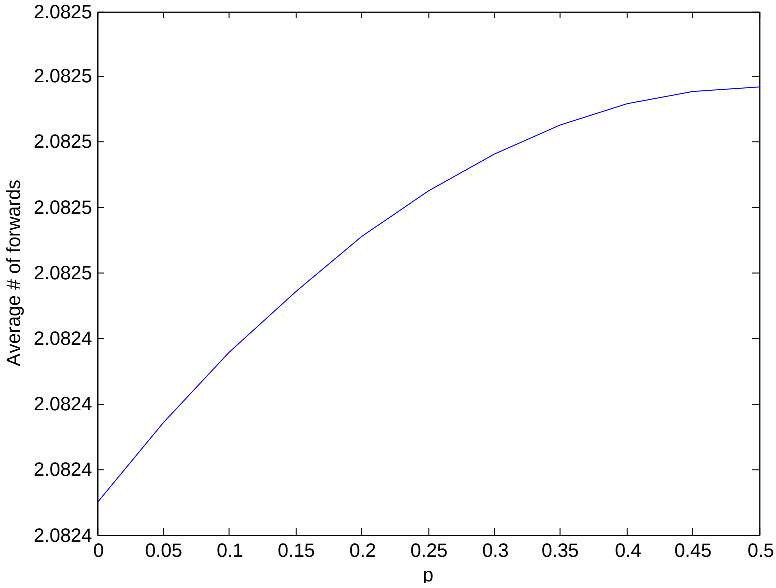



Figure 14: Performance of Random Left/Right with load= 0.8

As shown in figure 15, the performance gain of this algorithm is up to 5% for a ring size of $N = 10$. The list of coprimes in that scenario is $1, 3, 7, 9$, thus 4 possible choices. When increasing the ring size to 11, a prime number, the list of coprimes expands to 1..10 (because 11 is prime), thus 10 possible choices. This increases the relative performance gain up to 8%. To make clear the increased performance is not due to the increase of $N$, figure 17 shows the performance

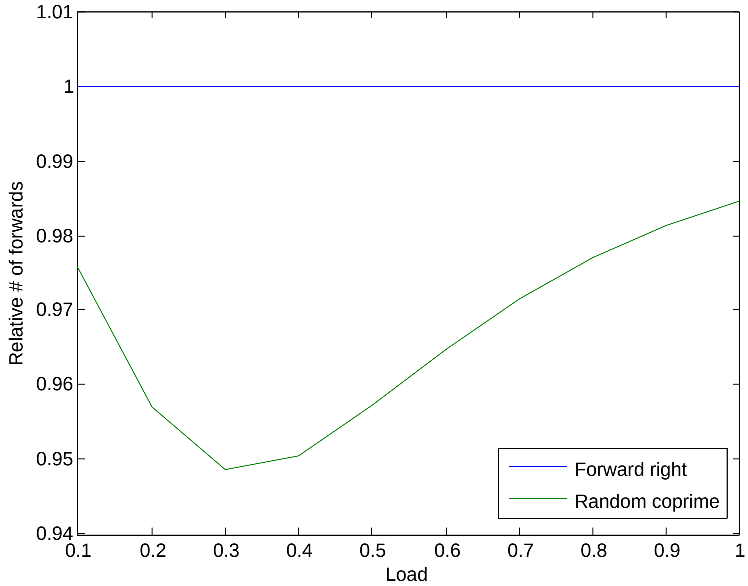



Figure 15: Validation of Coprime algorithm for $N = 10$

gain for $N = 12$.

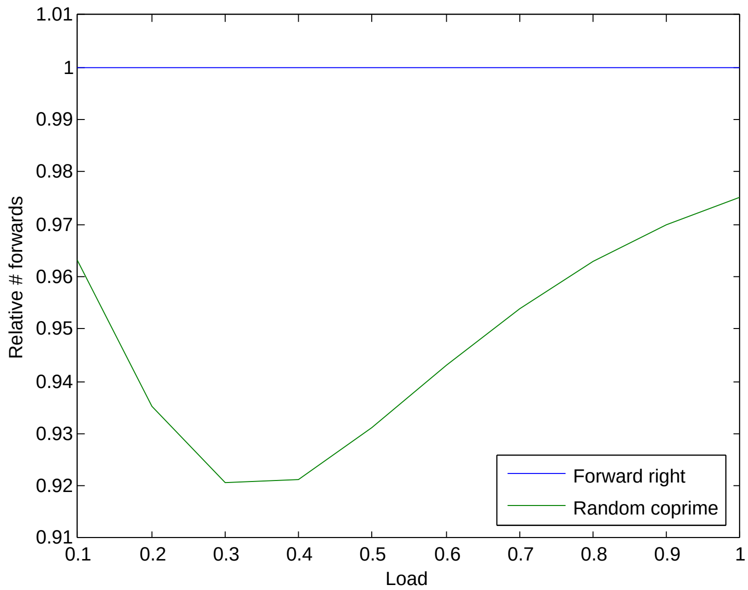



Figure 16: Validation of Coprime algorithm for $N = 11$

The performance gain is dependant on the number of different paths a job can follow. For $N = 10$ and $N = 12$, a job can follow 4 possible routes, for $N = 11$, 10 different routes can be chosen.

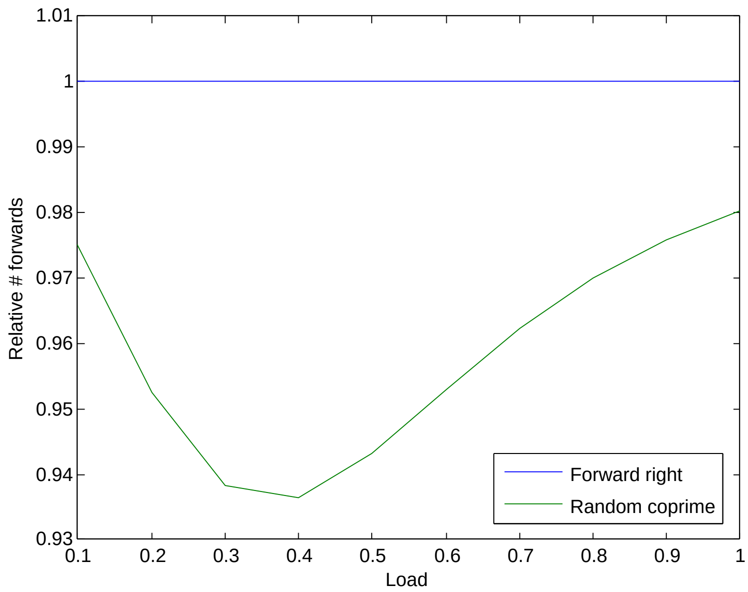



Figure 17: Validation of Coprime algorithm for $N = 12$

## 3.4 Random Unvisited

This problem can be modelled much more efficiently than the techniques. Since the next node is chosen at random, the information we need to save consists only of the number of servers which are currently busy. This problem is analogue to modelling an Erlang-B loss system. The number of states in this Markov Chain is linear to $N$, is much more dense and already represents a lumped Markov Chain. For $N = 3$, the matrix is given below.

$$
Q = \begin{array}{c c}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} &
\begin{bmatrix}
-3\lambda & 3\lambda & 0 & 0 \\
\mu & -3\lambda - \mu & 3\lambda & 0 \\
0 & \mu & -3\lambda - \mu & 3\lambda \\
0 & 0 & \mu & -\mu
\end{bmatrix}
\end{array}
$$

## 3.5 Lumped states

Except for Random Unvisited, each discribed technique is modelled into a Markov Chain with $N^2$ states. However, many of these states are redundant: for example, for $N = 3$ the states 001, 010 and 100 all represent one of the nodes being busy. For states representing multiple busy nodes, the space between these servers is critical information. Multiple states can be lumped when bitrotating one state can result in another state. Example: the states below are analogue and can therefore be lumped into one state:

`001101 = 011010 = 110100 = 101001 = 010011 = 100110`

The example model in 3.1 can be lumped into the following Markov Chain:

$$
Q = \begin{array}{c} \\ 000 \\ 001 \\ 011 \\ 111 \end{array}
\begin{array}{c}
\begin{array}{cccc} 000 & 001 & 011 & 111 \end{array} \\
\begin{bmatrix}
-3\lambda & 3\lambda & 0 & 0 \\
\mu & 3\lambda-\mu & 3\lambda & 0 \\
0 & 2\mu & 3\lambda-2\mu & 3\lambda \\
0 & 0 & 3\mu & -3\mu
\end{bmatrix}
\end{array}
$$

Computing the steady state distribution of a Markov Chain is subject to time constraints. Using sparse matrices for our algorithm already solved the memory constraints. Two factors are important when working with matrices: the number of elements and the number of nonzero elements. We will show that both factors are reduced significantly.

For unlumped Markov Chains modelling the Random forward algorithm, a matrix consists of $2^N$ states, an exponential growth. Lumping these matrices results in a number of states equal to: $\frac{1}{N}\sum_{d|N}(2^{N/d} \cdot \phi(d))$ where $\phi(d) = d \cdot \prod_{p|d, p \text{ is prime}}(1-\frac{1}{p})$ [1]. Although this result greatly reduces the number of states, its complexity is still non-polynomial.

The number of nonzero elements for unlumped Markov Chains is $(N+1)2^N$. For lumped matrices, we were not able to derive an exact formula, however, figure 19 shows a clear reduction as well. Yet, this result doesn't seem polynomial either.

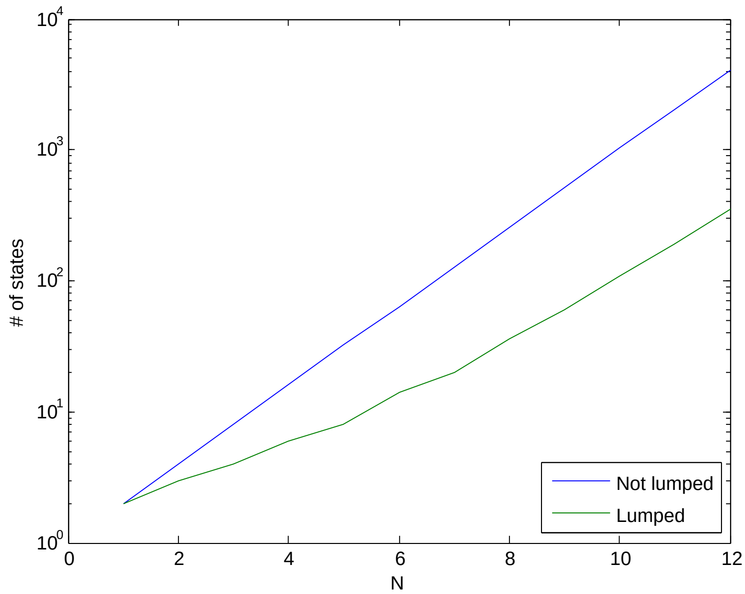



Figure 18: Number of states

It seems lumping is a good technique to push the bounderies of the validation by reducing two important factors of the compution time. However, it is no silver bullet: both the number of states and the number of nonzero elements are nonpolynomial after lumping the matrices.

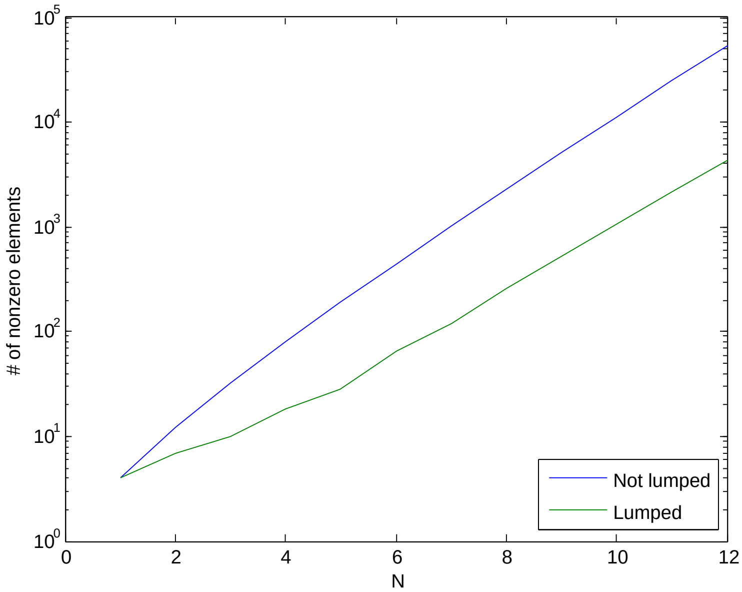



Figure 19: Number of nonzero elements

### 3.6 Equivalent techniques

Lumping states of a Markov Chain produces an equivalent Markov Chain. This can be used to prove some forwarding techniques are equal up to a certain $N$.

## 4 Conclusion

| | Max. gain | Space requirement (bits) | Notes |
|---|---|---|---|
| Forward Right | 0 (baseline) | 0 | |
| Left/Right forward | $> 4\%$ | 1 (job) + 1 (node) | |
| Random Left/Right forward | 0 (worse) | 1 (job) | |
| Position dependant forward | 0 (worse) | 1 (job) | Creates loaded clusters |

Table 4: Forward to neighbor

Which techniques work best in which environments? Why? Runner up? Why do some techniques don't work as expected? performance results in function of number of paths

## References


[1] The Online Encyclopedia of Integer Sequences. *A000031*. June 2009. URL: https://oeis.org/A000031.

| | Max. gain | Space requirement (bits) |
|---|---|---|
| Random unvisited | $> 24\%$ | $N$ |
| Coprime offset | $> 24\%$ | $log_2 N$ (job) + $log_2(\sum_{p<N,gcd(p,N)=1} 1)$ (node) |
| Random Coprime offset | $> 24\%$ | $log_2 N$ |

Table 5: Forward anywhere

| | Performance gain at load=0.5 | Number of possible paths |
|---|---|---|
| Forward Right | 0 | 1 |
| Left/Right forward | $\approx 4.5\%$ | 2 |
| Random Left/Right (0.5) | no gain | 2 |
| Position-dependant forwarding | no gain | 1 |
| Random unvisited | $\approx 24\%$ | $(N-1)!$ |
| (Random) Coprime offset | $\approx 24\%$ | $\sum_{p<N,gcd(p,N)=1} 1$ |

Table 6: Performance gain vs number of paths


[2] *The OpenMP® API specification for parallel programming.* URL: http://openmp.org/wp/.


# A Simulator source code

```
/*
 * nodes.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef NODES_H_
#define NODES_H_

#include "ring/node.h"
#include "ring/ring.h"
#include "ring/job.h"
#include <set>

struct DirectionInfo: public pop::JobInfo{
    inline DirectionInfo(double length):
        fLength(length), fDirection(0), fFirst(0)
    {}

    double fLength;
    short fDirection;
    pop::Node* fFirst;
};

struct VisitedInfo: public DirectionInfo{
```

```
    inline VisitedInfo(double length):
        DirectionInfo(length)
    {}

    std::set<unsigned int> visited;
};

class RightNode: public pop::Node {
public:
    typedef DirectionInfo info_type;

    RightNode(unsigned int id, pop::Ring* ring, unsigned int size);

    virtual ~RightNode(){}

    bool pushJob(pop::Job* j);
    void clearJob(pop::Job* j);

    bool wasHereFirst(pop::Job* j);
    bool accept(pop::Job* j);
};

class SwitchNode: public RightNode {
public:
    typedef DirectionInfo info_type;

    inline SwitchNode(unsigned int id, pop::Ring* ring, unsigned int size):
            RightNode(id, ring, size), last(1)
    {}

    bool pushJob(pop::Job* j);

protected:
    int last;
};

class RandSwitchNode: public RightNode {
public:
    typedef DirectionInfo info_type;

    inline static void setValue(double nv){
        v = nv;
    }

    inline RandSwitchNode(unsigned int id, pop::Ring* ring, unsigned int size):
            RightNode(id, ring, size)
    {}

    bool pushJob(pop::Job* j);

private:
    static double v;
};

class EvenSwitchNode: public RightNode{
public:
    typedef DirectionInfo info_type;

    inline EvenSwitchNode(unsigned int id, pop::Ring* ring, unsigned int size):
            RightNode(id, ring, size)
    {}

    bool pushJob(pop::Job* j);
};

class PrimeNode: public SwitchNode {
public:
    typedef DirectionInfo info_type;

    static void makePrimes(unsigned int size);

    inline PrimeNode(unsigned int id, pop::Ring* ring, unsigned int size):
        SwitchNode(id, ring, size)
    {
```

```
        if (fPrimes == 0){
            makePrimes(ring->getSize());
        }
    }

    bool pushJob(pop::Job* j);

protected:
    static int* fPrimes;
    static int fPrimesLen;
};

class RandPrimeNode: public PrimeNode{
public:
    typedef DirectionInfo info_type;

    RandPrimeNode(unsigned int id, pop::Ring* ring, unsigned int size):
        PrimeNode(id, ring, size)
    {
        if (fPrimes == 0){
            makePrimes(ring->getSize());
        }
    }

    bool pushJob(pop::Job* j);
};

class RandUnvisited: public RightNode{
public:
    typedef VisitedInfo info_type;

    RandUnvisited(unsigned int id, pop::Ring* ring, unsigned int size):
        RightNode(id, ring, size)
    {}

    bool pushJob(pop::Job* j);
};

class ToTopNode: public RightNode{
public:
    typedef DirectionInfo info_type;

    ToTopNode(unsigned int id, pop::Ring* ring, unsigned int size):
        RightNode(id, ring, size)
    {}

    bool pushJob(pop::Job* j);
};

class RRUnvisited: public RightNode{
public:
    typedef VisitedInfo info_type;

    RRUnvisited(unsigned int id, pop::Ring* ring, unsigned int size):
        RightNode(id, ring, size), offset(0)
    {}

    bool pushJob(pop::Job* j);

private:
    unsigned int offset;
};

#endif /* NODES_H_ */
```

Listing 2: nodes.h

```
/*
 * servernode.cpp
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

```

```
#include "servernode.h"

namespace pop {

ServerNode::ServerNode(unsigned int id, Ring* ring):
    Node(id, ring)
    {}

ServerNode::~ServerNode() {
    // TODO Auto-generated destructor stub
}

} /* namespace pop */
```

Listing 3: servernode.cpp

```
/*
 * configuration.h
 *
 *  Created on: Oct 6, 2011
 *      Author: ibensw
 */

#ifndef CONFIGURATION_H_
#define CONFIGURATION_H_

#include "ring/node.h"
#include "ring/job.h"

void help();

typedef pop::Node* (*makeNodeType)(unsigned int i, pop::Ring* r, unsigned int size);
typedef pop::JobInfo* (*makeInfoType)(double length);
struct Configuration{
    unsigned int seed;
    double joblength;
    double arrival;
    long nodes;
    unsigned int nodeSize;
    long progressinterval;
    long length;
    long repeat;
    makeNodeType makeNodeFunction;
    makeInfoType makeInfoFunction;

    Configuration(int argc, char** argv);
};

#endif /* CONFIGURATION_H_ */
```

Listing 4: randswitchchain.m

```
/*
 * nodes.cpp
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#include "nodes.h"
#include "ring/job.h"
#include "ring/ring.h"
#include "ring/finishevent.h"
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <string.h>

using namespace pop;
using namespace std;

RightNode::RightNode(unsigned int id, Ring* ring, unsigned int size):
    Node(id, ring, size)
{}
```

```
bool RightNode::wasHereFirst(Job* j){
    info_type* ji = dynamic_cast<info_type*>(j->getInfo());

    if (ji->fFirst == 0){
        ji->fFirst = this;
        return false;
    }

    return (ji->fFirst == this);
}

bool RightNode::accept(Job* j){
    if (!isBusy()){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        fCurrents.insert(j);
        double len = ji->fLength;
        fRing->getSimulator()->addEvent(new FinishEvent(fRing->getSimulator()->getTime()+
            len, j));
        return true;
    }
    return false;
}

bool RightNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        j->forward(fRing->getNode(this->fId+1));
    }

    return true;
}

void RightNode::clearJob(Job* j){
    fCurrents.erase(j);
}

bool SwitchNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            ji->fDirection = last;
            last*=-1;
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
    }
    return true;
}

double RandSwitchNode::v = 0.5;

bool RandSwitchNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            double rnd = (double)rand() / (double)RAND_MAX;
            ji->fDirection = (rnd < v ? 1 : -1);
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
    }
    return true;
```

```
}

bool EvenSwitchNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            ji->fDirection = ((this->getId() % 2 == 1) ? 1 : -1);
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
    }
    return true;
}

int* PrimeNode::fPrimes = 0;
int PrimeNode::fPrimesLen = 0;

unsigned int gcd(unsigned int a, unsigned b){
    unsigned int t;
    while(b){
        t=b;
        b=a%b;
        a=t;
    }
    return a;
}

void PrimeNode::makePrimes(unsigned int size){
    vector<unsigned int> primes;
    for (unsigned int i = 1; i < size; ++i){
        if (gcd(size, i) == 1){
            cout << "RelPrime: " << i << endl;
            primes.push_back(i);
        }
    }
    fPrimesLen = primes.size();
    fPrimes = new int[fPrimesLen];
    memcpy(fPrimes, primes.data(), fPrimesLen * sizeof(unsigned int));
}

bool PrimeNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            ji->fDirection = fPrimes[last];
            ++last;
            last%=fPrimesLen;
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
    }
    return true;
}

bool RandPrimeNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            ji->fDirection = fPrimes[rand() % fPrimesLen];
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
```

```
    }
    return true;
}

bool RandUnvisited::pushJob(Job* j){
    info_type* ji = dynamic_cast<info_type*>(j->getInfo());
    if (ji->visited.count(this->getId())){
        return false;
    }

    if (!accept(j)){
        ji->visited.insert(this->getId());

        if (fRing->getSize() == ji->visited.size()){
            return false;
        }

        unsigned int next;
        if (5*ji->visited.size() > 4*fRing->getSize()){
            unsigned int x = rand() % (fRing->getSize() - ji->visited.size());
            next = x;

            for (set<unsigned int>::iterator it = ji->visited.begin(); it != ji->visited.
                end(); it++){
                if (*it <= next){
                    ++next;
                }
            }
        }else{
            do{
                next = rand() % fRing->getSize();
            }while(ji->visited.count(next));
        }

        j->forward(fRing->getNode(next));
    }
    return true;
}

bool ToTopNode::pushJob(Job* j){
    if (wasHereFirst(j)){
        return false;
    }

    if (!accept(j)){
        info_type* ji = dynamic_cast<info_type*>(j->getInfo());
        if (ji->fDirection == 0){
            if (this->getId() > fRing->getSize()/2){
                ji->fDirection = 1;
            }else{
                ji->fDirection = -1;
            }
        }

        j->forward(fRing->getNode(this->fId + ji->fDirection));
    }
    return true;
}

bool RRUnvisited::pushJob(Job* j){
    info_type* ji = dynamic_cast<info_type*>(j->getInfo());
    if (ji->visited.count(this->getId())){
        return false;
    }

    if (!accept(j)){
        ji->visited.insert(this->getId());

        if (fRing->getSize() == ji->visited.size()){
            return false;
        }

        unsigned int next = this->getId() + offset + 1;
        ++offset;
```

```
        offset%=(fRing->getSize()-1);

        next%=fRing->getSize();
        while (ji->visited.count(next)){
            next++;
            next%=fRing->getSize();
        }

        j->forward(fRing->getNode(next));
    }
    return true;
}
```

Listing 5: randswitchchain.m

```
#include <iostream>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#include "ring/ring.h"
#include "ring/job.h"
#include "ring/arriveevent.h"
#include "nodes.h"
#include "configuration.h"
using namespace pop;
using namespace std;

double exp_distr(double lambda){
    double r = (double)rand() / (double)RAND_MAX;
    return -lambda * log(r);
}

void preload(Configuration c, Ring* r){
    double rnd;
    double l;
    double load = c.length/c.arrival;
    for (unsigned int i = 0; i < r->getSize(); ++i){
        rnd = (double)rand() / (double)RAND_MAX;
        if (rand() < load){
            l=exp_distr(c.joblength);
            r->getSimulator()->addEvent(new ArriveEvent(0.0, new Job(c.makeInfoFunction(l
                )), r->getNode(i)));
        }
    }
}

void fillEvents(Configuration c, Ring* r){
    double t;
    double l;
    Node* n;
    for (unsigned int i = 0; i < r->getSize(); ++i){
        n=r->getNode(i);
        t = 0.0;
        while (t < c.length){
            t+=exp_distr(c.arrival);
            l=exp_distr(c.joblength);
            r->getSimulator()->addEvent(new ArriveEvent(t, new Job(c.makeInfoFunction(l))
                , n));
        }
    }
}


int main(int argc, char** argv) {
    Configuration c(argc, argv);

    double success = 0.0;
    double avghops = 0.0;

    cout.setf(ios::fixed,ios::floatfield);
    cout.precision(12);

#pragma omp parallel for
```

```
    for (unsigned int i = 0; i < c.repeat; ++i){
        Ring r(c.nodes, c.nodeSize, c.makeNodeFunction);

        //preload(c, &r); //disabled to compare output to more early results

        fillEvents(c, &r);

        if (c.progressinterval <= 0){
            r.getSimulator()->run();
        }else{
            r.getSimulator()->run(c.progressinterval);
        }

#pragma omp critical
        {
            cout << "---------------------------" << endl;
            cout << "Run: " << i << endl;
            cout << "Total jobs:\t\t" << r.getTotalJobs() << endl;
            cout << "Finished jobs:\t\t" << r.getFinishedJobs() << endl;
            cout << "Discarded jobs:\t\t" << r.getDiscardedJobs() << endl;
            cout << "Total hops (finished):\t" << r.getFinishedJobTotalHops() << endl;
            long totalhops = r.getFinishedJobTotalHops() + (c.nodes-1) * r.getDiscardedJobs();
            cout << "Total hops (all):\t" << totalhops << endl;
            cout << "Hops/job (finished):\t" << (double)r.getFinishedJobTotalHops()/r.getFinishedJobs() << endl;
            cout << "Hops/job (total):\t" << (double)totalhops/r.getTotalJobs() << endl;
            cout << "Success ratio:\t\t" << (100.0 * r.getFinishedJobs()) / r.getTotalJobs() << "%" << endl;
            success+=(double)(r.getFinishedJobs()) / r.getTotalJobs();
            avghops+=(double)r.getFinishedJobTotalHops()/r.getFinishedJobs();
        }
    }

    if (c.repeat > 1){
        cout << "---------------------------" << endl;
        cout << "Avg. hops/job (finished):\t" << avghops / c.repeat << endl;
        cout << "Avg. success ratio:\t\t" << 100.0 * success / c.repeat << "%" << endl;
    }

    return 0;
}
```

Listing 6: randswitchchain.m

```
/*
 * servernode.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef SERVERNODE_H_
#define SERVERNODE_H_

#include "ring/node.h"
#include "ring/ring.h"

namespace pop {

class ServerNode: public Node {
public:
    ServerNode(unsigned int id, Ring* ring);
    virtual ~ServerNode();
};

} /* namespace pop */
#endif /* SERVERNODE_H_ */
```

Listing 7: randswitchchain.m

```
/*
 * ring.cpp
 *
```

```
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#include "ring.h"

namespace pop {

Ring::Ring(unsigned int size, unsigned int nodesize, Node* (*mkNode)(unsigned int i, Ring
    * r, unsigned int ns)):
    fSize(size),
    fRing(new Node*[size]),
    jobsTotal(0), jobsFinished(0), jobsDiscarded(0), jobsFinishedTotalHops(0)
{
    for (unsigned int i = 0; i < size; ++i){
        fRing[i] = mkNode(i, this, nodesize);
    }
}

Ring::~Ring() {
    for (unsigned int i = 0; i < fSize; ++i){
        delete fRing[i];
    }
    delete[] fRing;
}

} /* namespace pop */
```

Listing 8: randswitchchain.m

```
/*
 * job.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef JOB_H_
#define JOB_H_

#include "node.h"

namespace pop {

class JobInfo {
public:
    inline JobInfo(){}

    virtual ~JobInfo(){}
};

class Job {
public:
    Job(JobInfo* ji);
    virtual ~Job();

    inline Node* getCurrentNode(){
        return fCurrent;
    }

    inline JobInfo* getInfo(){
        return fJobInfo;
    }

    void forward(Node* n);
    void finish(double time);
    void discard();

private:
    double fStart;
    double fFinish;
    Node* fCurrent;
    unsigned int fHops;
    JobInfo* fJobInfo;
```

```
};

} /* namespace pop */
#endif /* JOB_H_ */
```

Listing 9: randswitchchain.m

```
/*
 * node.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef NODE_H_
#define NODE_H_

#include <set>

namespace pop {
class Ring;
class Job;

class Node {
public:
    Node(unsigned int id, Ring* ring, unsigned int size = 1);
    virtual ~Node();

    inline unsigned int getId() const{
        return fId;
    }

    inline Ring* getRing() const {
        return fRing;
    }

    inline unsigned int getTotalSize() const{
        return fSize;
    }

    inline unsigned int getSize() const{
        return fCurrents.size();
    }

    inline bool isBusy() const{
        return fCurrents.size() == fSize;
    }

    virtual bool pushJob(Job* j) = 0;
    virtual void clearJob(Job* j) = 0;

protected:
    unsigned int fId;
    Ring* fRing;
    std::set<Job*> fCurrents;
    unsigned int fSize;
};

} /* namespace pop */
#endif /* NODE_H_ */
```

Listing 10: randswitchchain.m

```
/*
 * ring.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef RING_H_
#define RING_H_

#include "node.h"
```

```
#include "../simulator/simulator.h"

//#include <iostream>

namespace pop {

class Ring {
public:
    Ring(unsigned int size, unsigned int nodesize, Node* (*mkNode)(unsigned int i, Ring*
        r, unsigned int ns));
    virtual ~Ring();

    inline unsigned int getSize(){
        return fSize;
    }

    inline Node* getNode(int id){
        //std::cout << "id: " << id << " = " << (id + fSize) % fSize << std::endl;
        return fRing[(id + fSize) % fSize];
    }

    inline Simulator* getSimulator(){
        return &fSimulator;
    }

    inline unsigned int getTotalJobs(){
        return jobsTotal;
    }
    inline unsigned int getDiscardedJobs(){
        return jobsDiscarded;
    }
    inline unsigned int getFinishedJobs(){
        return jobsFinished;
    }
    inline unsigned int getFinishedJobTotalHops(){
        return jobsFinishedTotalHops;
    }

    inline void jobCreated(){
        ++jobsTotal;
    }

    inline void jobFinished(unsigned int hops){
        ++jobsFinished;
        jobsFinishedTotalHops+=hops;
    }

    inline void jobDiscarded(){
        ++jobsDiscarded;
    }

private:
    unsigned int fSize;
    Node** fRing;
    Simulator fSimulator;

    unsigned int jobsTotal;
    unsigned int jobsFinished;
    unsigned int jobsDiscarded;
    unsigned int jobsFinishedTotalHops;
};

} /* namespace pop */
#endif /* RING_H_ */
```

Listing 11: randswitchchain.m

```
/*
 * node.cpp
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

```

```
#include "node.h"

namespace pop {

Node::Node(unsigned int id, Ring* ring, unsigned int size):
    fId(id), fRing(ring), fSize(size)
    {}

Node::~Node() {
    // TODO Auto-generated destructor stub
}

} /* namespace pop */
```

Listing 12: randswitchchain.m

```
/*
 * events.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef EVENTS_H_
#define EVENTS_H_

#include "../simulator/event.h"
#include "job.h"

namespace pop {

class ArriveEvent : public Event {
public:
    inline ArriveEvent(double scheduled, Job* job, Node* n):
        Event(scheduled), j(job), first(n)
    {}

    inline void run(Simulator* simulator){
        j->forward(first);
        delete this;
    }

private:
    Job* j;
    Node* first;
};

} /* namespace pop */
#endif /* EVENTS_H_ */
```

Listing 13: randswitchchain.m

```
/*
 * job.cpp
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#include "job.h"
#include "ring.h"
#include <iostream>
using namespace std;

namespace pop {

Job::Job(JobInfo* ji):
    fStart(-1.0), fFinish(-1.0), fCurrent(0), fHops(-1), fJobInfo(ji)
    {}

Job::~Job() {
    if (fJobInfo){
        delete fJobInfo;
    }
```

```
}

void Job::discard(){
    fCurrent->getRing()->jobDiscarded();
    //cout << fCurrent->getId() << "\tJob discarded\t(arrival time: " << fStart << " / #
        hops: " << fHops << ")" << endl;
    delete this;
}

void Job::finish(double time){
    fCurrent->getRing()->jobFinished(fHops);
    fFinish = time;
    //cout << fCurrent->getId() << "\tJob finished\t(arrival time: " << fStart << " /
        finish time: " << fFinish << " / #hops: " << fHops << ")" << endl;
    fCurrent->clearJob(this);
    delete this;
}

void Job::forward(Node* n){
    if (!fCurrent){
        n->getRing()->jobCreated();
        fStart = n->getRing()->getSimulator()->getTime();
    }
    ++fHops;
    fCurrent = n;
    if (!n->pushJob(this)){
        discard();
    }
}

} /* namespace pop */
```

Listing 14: randswitchchain.m

```
/*
 * finishevent.h
 *
 *  Created on: Sep 27, 2011
 *      Author: ibensw
 */

#ifndef FINISHEVENT_H_
#define FINISHEVENT_H_

#include "../simulator/simulator.h"
#include "../simulator/event.h"
#include "job.h"

namespace pop {

class FinishEvent: public Event {
public:
    inline FinishEvent(double scheduled, Job* job):
        Event(scheduled), j(job)
    {}

    inline void run(Simulator* simulator){
        j->finish(simulator->getTime());
        delete this;
    }

private:
    Job* j;
};

} /* namespace pop */
#endif /* FINISHEVENT_H_ */
```

Listing 15: randswitchchain.m

```
#include <iostream>
#include <stdlib.h>
#include "configuration.h"
#include "nodes.h"
```

```
using namespace std;

void help(){
    cout << "Usage: -r -s long -j double -a double -n long -p long -l long -t long -h type" << endl;
    cout << "\t-r\tRandom seed" << endl;
    cout << "\t-s\tSet seed\t\t\t(default: 0)" << endl;
    cout << "\t-j\tJob length\t\t\t(default: 1.0)" << endl;
    cout << "\t-a\tLoad\t\t\t\t(default: 1.0)" << endl;
    cout << "\t-n\tRing size\t\t\t(default: 100)" << endl;
    cout << "\t-c\tProcessing units per node\t(default: 1)" << endl;
    cout << "\t-p\tPrint progress interval\t\t(default: -1 - disabled)" << endl;
    cout << "\t-l\tSimulation length\t\t(default: 3600)" << endl;
    cout << "\t-t\tRepetition\t\t\t(default: 1)" << endl;
    cout << "\t-h\tPrint this help" << endl;
    cout << "\t type\tright | switch | randswitch | evenswitch | prime | randprime | randunvisited | totop" << endl;
}

template <typename T>
pop::JobInfo* createJI(double len){
    return new T(len);
}

template <typename T>
pop::Node* createN(unsigned int id, pop::Ring* ring, unsigned int size){
    return new T(id, ring, size);
}

Configuration::Configuration(int argc, char** argv):
    seed(0), joblength(1.0),
    arrival(1.0), nodes(100), nodeSize(1), progressinterval(-1), length(3600),
    repeat(1), makeNodeFunction(0), makeInfoFunction(0)
{
    int c;
    int index;
    double load = 1.0;
    while ((c = getopt (argc, argv, "rs:j:a:n:c:p:l:hv:t:")) != -1){
        switch (c){
        case 'r':
            seed = time(0);
            break;
        case 's':
            seed = atol(optarg);
            break;
        case 'j':
            joblength = atof(optarg);
            break;
        case 'a':
            load = atof(optarg);
            break;
        case 'n':
            nodes = atol(optarg);
            break;
        case 'c':
            nodeSize = atol(optarg);
            break;
        case 'p':
            progressinterval = atoi(optarg);
            break;
        case 'l':
            length = atol(optarg);
            break;
        case 't':
            repeat = atol(optarg);
            break;
        case 'h':
            help();
            exit(0);
            break;
        case 'v':
            RandSwitchNode::setValue(atof(optarg));
            break;
```

```
        default:
            cout << "Unknown option: " << optopt << endl;
            break;
        }
    }

    arrival = joblength/load/nodeSize;

    srand(seed);
    cout << "Seed: " << seed << endl
         << "Interarrival time: " << arrival << endl;

    for (index = optind; index < argc; index++){
        string arg = argv[index];
        if (arg == "right"){
            makeNodeFunction = createN<RightNode>;
            makeInfoFunction = createJI<RightNode::info_type>;
        }
        if (arg == "switch"){
            makeNodeFunction = createN<SwitchNode>;
            makeInfoFunction = createJI<SwitchNode::info_type>;
        }
        if (arg == "randswitch"){
            makeNodeFunction = createN<RandSwitchNode>;
            makeInfoFunction = createJI<RandSwitchNode::info_type>;
        }
        if (arg == "evenswitch"){
            makeNodeFunction = createN<EvenSwitchNode>;
            makeInfoFunction = createJI<EvenSwitchNode::info_type>;
        }
        if (arg == "prime"){
            makeNodeFunction = createN<PrimeNode>;
            makeInfoFunction = createJI<PrimeNode::info_type>;
        }
        if (arg == "randprime"){
            makeNodeFunction = createN<RandPrimeNode>;
            makeInfoFunction = createJI<RandPrimeNode::info_type>;
        }
        if (arg == "randunvisited"){
            makeNodeFunction = createN<RandUnvisited>;
            makeInfoFunction = createJI<RandUnvisited::info_type>;
        }
        if (arg == "totop"){
            makeNodeFunction = createN<ToTopNode>;
            makeInfoFunction = createJI<ToTopNode::info_type>;
        }
        if (arg == "rrunvisited"){
            makeNodeFunction = createN<RRUnvisited>;
            makeInfoFunction = createJI<RRUnvisited::info_type>;
        }
    }

    if (!makeNodeFunction){
        cerr << "No type given" << endl;
        exit(1);
    }
}
```

Listing 16: randswitchchain.m

```
/*
 * schedule.h
 *
 *  Created on: Sep 26, 2011
 *      Author: ibensw
 */

#ifndef SCHEDULE_H_
#define SCHEDULE_H_

#include "event.h"

namespace pop {

```

```
class Schedule {
public:
    inline Schedule(Event* e):
        fE(e)
    {}

    inline bool operator<(const Schedule& s) const{
        return fE->getScheduleTime() > s.fE->getScheduleTime();
    }

    inline Event* getEvent(){
        return fE;
    }

private:
    Event* fE;
};

} /* namespace pop */
#endif /* SCHEDULE_H_ */
```

Listing 17: randswitchchain.m

```
/*
 * event.h
 *
 *  Created on: Sep 26, 2011
 *      Author: ibensw
 */

#ifndef EVENT_H_
#define EVENT_H_

//#include "simulator.h"

namespace pop {
class Simulator;

class Event {
public:
    inline Event(double scheduled):
        fScheduled(scheduled)
        {}

    inline virtual ~Event(){

    }

    inline double getScheduleTime(){
        return fScheduled;
    }

    virtual void run(Simulator* simulator) = 0;

protected:
    double fScheduled;
};

} /* namespace pop */
#endif /* EVENT_H_ */
```

Listing 18: randswitchchain.m

```
/*
 * simulator.h
 *
 *  Created on: Sep 26, 2011
 *      Author: ibensw
 */

#ifndef SIMULATOR_H_
#define SIMULATOR_H_

#include <queue>
```

```
#include "event.h"
#include "schedule.h"

namespace pop {

class Simulator {
public:
    Simulator();
    virtual ~Simulator();

    void run();
    void run(int infointerval);

    inline double getTime(){
        return fNow;
    }

    inline unsigned int getPendingEvents(){
        return fPending.size();
    }

    void addEvent(Event* e);

private:
    std::priority_queue<Schedule> fPending;
    double fNow;
};

} /* namespace pop */
#endif /* SIMULATOR_H_ */
```

Listing 19: randswitchchain.m

```
/*
 * simulator.cpp
 *
 *  Created on: Sep 26, 2011
 *      Author: ibensw
 */

#include "simulator.h"
#include <iostream>
using namespace std;

namespace pop {

Simulator::Simulator():
    fNow(0.0)
        {}

Simulator::~Simulator() {
}

void Simulator::run(){
    while(!fPending.empty()){
        Schedule x = fPending.top();
        fPending.pop();
        fNow = x.getEvent()->getScheduleTime();
        x.getEvent()->run(this);
    }
}

void Simulator::run(int interval){
    int next = interval;
    while(!fPending.empty()){
        Schedule x = fPending.top();
        fPending.pop();
        fNow = x.getEvent()->getScheduleTime();
        if (fNow > next){
            cout << "Time: " << fNow << "\tPending events: " << fPending.size() << endl;
            next+=interval;
        }
        x.getEvent()->run(this);
    }
```

```
}

void Simulator::addEvent(Event* e){
    fPending.push(Schedule(e));
}

}
```

Listing 20: randswitchchain.m

```
/*
 * schedule.cpp
 *
 *  Created on: Sep 26, 2011
 *      Author: ibensw
 */

#include "schedule.h"

namespace pop {


} /* namespace pop */
```

Listing 21: randswitchchain.m

# B MATLAB Numerical evaluation code

```
function [Q] = rightchain(size, rate)
%RIGHTCHAIN Generate a Markov Chain that always forwards right
%Parameters:
%        size     The size of the ring
%        rate     The rate of arrivals

    totalsize = 2^size;
    Q = sparse(totalsize, totalsize);

    BITS = zeros(1, size);

    for i=1:size
        BITS(i) = 2^(i-1);
    end

    for i=0:(totalsize-1)
        t=0;
        for b=1:size
            j=bitxor(i, BITS(b));
            if bitand(i, BITS(b))
                Q(i+1, j+1)=1;
            else
                r=rate;
                bt=b+1;
                while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (b))
                    bt=bt+1;
                    r = r + rate;
                end
                Q(i+1, j+1)=r;
            end
            t=t + Q(i+1, j+1);
        end
        Q(i+1, i+1) = -t;
    end

end
```

Listing 22: rightchain.m

```
function [Q] = randswitchchain(size, rate, p)
%RANDSWITCHCHAIN Generates a Markov Chain that randomly forward left or right
%Parameters:
```

```
%        size     The size of the Markov Chain
%        rate     The rate of arrivals
%        p                The probability a job is forwarded right
%                         (Default: 0.5)

         if nargin < 3
                 p=0.5;
         end

    totalsize = 2^size;
    Q = sparse(totalsize, totalsize);

    BITS = 2.^[0:size-1];

    for i=0:(totalsize-1)
         t=0;
         for b=1:size
              j=bitxor(i, BITS(b));
              if bitand(i, BITS(b))
                  Q(i+1, j+1)=1;
              else
                  r=rate;
                  bt=b+1;
                  while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (b))
                       bt=bt+1;
                       r = r + rate*p;
                  end
                  bt=b-1;
                  while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (b))
                       bt=bt-1;
                       r = r + rate*(1-p);
                  end
                  Q(i+1, j+1)=r;
              end
              t=t + Q(i+1, j+1);
         end
         Q(i+1, i+1) = -t;
    end

end
```

Listing 23: randswitchchain.m

```
function [ Q ] = rprimechain( size, rate )
%RPRIMECHAIN Generate a Markov Chain that chooses a random coprime and uses this as
    forwarding offset
%Parameters:
%        size     The size of the ring
%        rate     The arrival rate

         totalsize=2^size;
         rprimes=[];

         for i=1:(size-1)
                 if gcd(size, i) == 1
                         rprimes=[rprimes i];
                 end
         end

         rpcount = length(rprimes);

         %Q=zeros(totalsize);
         Q=sparse(totalsize, totalsize);

         for i=0:totalsize-1
                 tot=0;
                 for j=0:size-1
                         k=2^j;
                         if bitand(i,k)
                                 Q(i+1, i-k+1) = 1.0;
                                 tot=tot+1.0;
                         else
                                 c=0;
                                 for p=rprimes
```

```
							current=mod(j-p, size);
							while (bitand(i,2^current))
								current=mod(current-p, size);
								c=c+1;
							end
						end
						Q(i+1, i+k+1) = rate + c*rate/rpcount;
						tot=tot+Q(i+1, i+k+1);
					end
				end
				Q(i+1, i+1) = -tot;
	end

end
```

Listing 24: rprimechain.m

```
function [ Q ] = runvisitedchain( size, rate )
%RUNVISITEDCHAIN Generate a Markov Chain that forwards to an unvisited node
%Parameters:
%	size	The size of the ring
%	rate	The arrival rate

	rate = rate*size;

	Q = sparse(size+1, size+1);

	Q(1,2) = rate;
	Q(1,1) = -rate;

	Q(size+1, size) = size;
	Q(size+1, size+1) = -size;

	for i=2:size
		Q(i,i-1) = i-1;
		Q(i,i+1) = rate;
		Q(i,i) = -Q(i,i+1)-Q(i,i-1);
	end

end
```

Listing 25: runvisitedchain.m

```
function [ avg,distribution ] = avghops(Q, d)
%AVGHOPS Calculate average number of times a job is forwarded
%Parameters:
%	Q	The matrix representing a markov chain
%Optional:
%	d	Debug mode, default=1, disable debug output=0
%Return:
%	avg	The average number of forwards
%	distribution	The distribution for each possible outcome

	if nargin < 2
		d=1;
	end

	steady=full(ctmcsteadystate(Q));
	distribution=zeros(1,d);

	len=length(Q);
	states=log2(len);
	avg=0;
	total=0;

	for i=0:(states-1)
		c=0;
		prefix=((2^i)-1) * 2^(states-i);
		for j=0:(2^(states-i-1))-1
			c=c+steady(prefix + j + 1);
		end
		total=total+c;
		if d
			fprintf('%d hops:\t%f\n', i, c);
```

```
            end
            distribution(i+1)=c;
            avg=avg+(c*i);
        end

        loss=steady(len);
        avg=avg/(1-loss);
        if d
            fprintf('Loss:\t%f\nTotal:\t%f\nAverage␣#hops:\t%f\n', loss, total + loss
                , avg);
        end
end
```

Listing 26: avghops.m

```
function [ avg ] = ruavghops( Q, d )
%RUAVGHOPS Calculate average number of times a job is forwarded for the random unvisited
    chain
%Parameters:
%       Q       The matrix representing a markov chain using the random unvisited
    forwarding algorithm
%Optional:
%       d       Debug mode, default=1, disable debug output=0

        if nargin < 2
            d=1;
        end

        steady=ctmcsteadystate(Q);

        len=length(Q);

        avg = 0;
        avgp = zeros(1, len);

        for i=0:len-2
            tmpavg = 0;
            for h=0:i
                c = prod(i-h+1:i) * (len-1-i) / prod(len-1-h:len-1);
                tmpavg = tmpavg + (c * h);
                avgp(h+1) = avgp(h+1) + (c*steady(i+1));
            end
            avg=avg + steady(i+1) * tmpavg;
        end

        avgp(len) = steady(len);

        loss=steady(len);
        avg=avg/(1-loss);
        if d
            avgp
            fprintf('Loss:\t%f\nAverage␣#hops:\t%f\n', loss, avg);
        end

end
```

Listing 27: ruavghops.m

```
function [ pi ] = ctmcsteadystate( Q )
%CTMCSTEADYSTATE Steady state distribution of a continious time markov chain
%Parameters:
%       Q       Matrix representing a Markov Chain
%Source: http://speed.cis.nctu.edu.tw/~ydlin/course/cn/nsd2009/Markov-chain.pdf (slide
    10)

        T=Q;
        len=length(Q);
        T(:,len)=ones(len, 1);
        e=zeros(1, len);
        e(len)=1;
        pi=e*inv(T);
end
```

Listing 28: ctmcsteadystate.m

```
function [ avg ] = lumpavghops(Q)
%LUMPAVGHOPS Get the average number of times a job is forwarded when the state matrix is
     lumped
%Parameters:
%       Q       A lumped matrix representation of a markov Chain

        fullsize=length(Q);
        [Q S]=lump(Q);
        lumpsize=length(S);
        nodes=log2(fullsize);
        hops=zeros(1,nodes+1);

        steady=ctmcsteadystate(Q);

        hops(1)=steady(1); %zero hops
        hops(nodes+1)=steady(lumpsize); %loss
        for i=2:lumpsize-1;
                bits=ceil(log2(S(i)+1));
                hops(1)=hops(1)+(nodes-bits)/nodes*steady(i);

                for j=bits:-1:1
                        c=0;
                        while c<j && bitand(S(i), 2^(j-c-1))
                                c=c+1;
                        end
                        hops(c+1)=hops(c+1) + steady(i)/nodes;
                end
        end

        %fprintf('Sum:\t%f\n',sum(hops));

        avg=(hops(1:nodes)*[0:nodes-1]')/(1-steady(lumpsize));

end
```

Listing 29: lumpavghops.m

```
function [Ql S] = lump(Q)
%LUMP Lump a matrix representing a Markov Chain
%Parameters:
%       Q       The matrix that should be lumped
%Return:
%       Ql      The lumped matrix representation
%       S       The states that are used in the lumped matrix
%The states of the matrix Q must represent the availability of the the servers

        [S R C] = makestates(log2(length(Q)));

        Ql=sparse(length(S), length(S));

        [i j s] = find(Q);

        for x=1:length(i)
                Ql(R(i(x)),R(j(x)))=Ql(R(i(x)),R(j(x)))+s(x);
        end

        for x=1:length(S)
                Ql(x,:)=Ql(x,:)/C(x);
        end

end
```

Listing 30: lump.m

```
function [r, refindex, coverage] = makestates(rsize)
%Generate lumped states
%Parameters:
%       rsize           Size of the ring (or log2 of the number of states of the matrix)
%Return:
%       r               Vector of the remaining states, ordered
%       refindex        Reference index, each old state points to the new lumped state
%       coverage        How many states the lumped state with the same index represents

```

```
    powers = 2.^[0:rsize-1];

    function [v] = rotate(a, size)
        p=2^(size-1);
        v = a*2 + floor(a/p) - 2*p*floor(a/p);
    end

    function [r] = makesmallest(a)
        r=a;
        for i=1:(rsize-1)
            a=rotate(a,rsize);
            if a<r
                r=a;
            end
        end
    end

    refindex = [];
    for i=0:(2^rsize)-1
        refindex = [refindex makesmallest(i)];
    end

    function [c] = cover(a, size)
        c=1;
        a=makesmallest(a);
        b=rotate(a, size);
        while a ~= b
            b=rotate(b, size);
            c=c+1;
        end
    end

    function [r] = smallest(a, size)
        r=a;
        for i=1:(size-1)
            a=rotate(a,size);
            if a<r
                r=a;
            end
        end
    end

    function [v] = f(word, bits, place, size)
        if place > size
            v = word;
        elseif bits == 0
            v = f(word, bits, place+1, size);
        elseif place + bits > size
            v = f(word + powers(place), bits-1, place+1, size);
        else
            v = [f(word + powers(place), bits-1, place+1, size) f(word, bits,
                place+1, size)];
        end
    end


    function [r] = makecombs(k, n)
        leadzeros = ceil(n/k)-1;
        fullsize = n - leadzeros - 1;
        r = f(0,k-1,1,fullsize)*2 + 1;
    end

    r=[0 2^(rsize)-1];
    for i=1:rsize-1
        r = [r makecombs(i, rsize)];
    end

    s=[];
    for i=r
        s=[s refindex(i+1)];
    end

    r=unique(s);
    refindex = arrayfun(@(x) find(r == x), refindex);
```

```

        coverage=[];
        for w=r
                coverage=[coverage cover(w, rsize)];
        end

end
```

Listing 31: makestates.m