A Graph Parallel Implementation of Hidden Markov Models

Kevin Schmid, Andy Shi, Ding Zhou
CS 205
Harvard College
Cambridge, MA

May 11, 2015

1 Introduction

Hidden Markov models (HMM) are often used to model time-series data for both discrete and continuous data. HMM has been successfully applied to problems such as speech recognition [1] and gesture recognition [2]. The parameters to a hidden Markov models are most commonly learned from the Baum-Welch—an Expectation Maximization (EM) algorithm. Unfortunately, the Baum-Welch algorithm is quadratic in number of states. A parallel implementation of the EM algorithm along the state axis would greatly improve scalability. Graph based platforms are particularly suited to EM algorithms, and our project uses GraphLab [3], a platform created by Dato [4] for implementing graph based parallelizing schemes.

Our code is available at `https://github.com/cs205-project-group/hmm`.

2 Background

2.1 Hidden Markov Model

A hidden Markov model assumes that the system follows a Markov process. For this paper, we will assume a finite number of states and observations. The probability of transitioning from state i to state j only depends on the current state, and we define transition matrix A such that element a_{ij} is the probability of transitioning from i to j. In an HMM, we cannot observe states, but we can observe outputs generated by the states. Given N states and M observations, we capture this with a $N \times M$ emission matrix B, where each element b_{ij} is the probability of observation j given the state is i. Finally, we define a N length vector as π as the prior, where π_i is the probability of initial state being i.

We can thus capture all the information about an HMM with $\theta = (A, B, \pi)$. In an HMM, we are given a sequence of T observations, and we will use a series of forward and backwards propagations to estimate the parameters $\theta = (A, B, \pi)$.

2.2 Baum-Welch

The following explanation is based on Wikipedia's presentation of the algorithm [5].

Let $(X_1, X_2, \ldots, X_N)$ denote the N states and $\mathbf{O} = \{O_1, O_2, \ldots, O_T\}$ the T observed outputs. The Baum-Welch algorithm uses expectation-maximization to find the maximum likelihood estimate.

Suppose we begin with some initial parameters, $\theta = (A, B, \pi)$. Given the observation sequence, our task is to update these parameters based on insights from the training sequence.

Given our training sequence, Baum-Welch first figures out the probability of being in a state at some time in the "run" of the observation sequence. Let's call this $\gamma_i(t)$, the "probability of being in state i at time t of this observation sequence given our current model parameters," $P(X_t = i|\mathbf{O}, \theta)$.

How can we figure this out? To do this, the Baum-Welch algorithm performs two subcalculations, for the "forward probabilities" and the "backward probabilities." Like $\gamma_i(t)$ values, the forward probability are computed for each (*state*, *time*) pair: given some observation sequence and our current moel, the forward probability $\alpha_i(t)$ is the probability of seeing the observation sequence up to time t and landing in state i. The backward probability is defined analogously: it's the probability of starting in some state i and seeing the observation sequence *from* time t until the end.

We can define the forward and backward probabilities recursively:

$$\alpha_i(t+1) = b_{jo_1} \sum_{i=1}^{N} \alpha_i(t) a_{ij} \tag{1}$$

$$\beta_i(t) = \sum_{i=1}^{N} \beta_j(t+1) a_{ij} b_j(o_{t+1}) \tag{2}$$

We can put the alpha and beta probabilities together to find $\gamma_i(t)$. Some intuition for this formula: the bottom is the probability of being in *some state* at time t given our *entire* observation sequence. The top is the probability of being in some particular state i at time t given our *entire* observation sequence. For both numerator and denominator, you have to get to the state at that time (forward probability) and leave from the state next and finish emitting the observation sequence (backward probability).

$$\gamma_i(t) = P(X_t = i | \mathbf{O}, \theta) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)} \tag{3}$$

From here, we can figure out our updates for the B matrix and the new prior.

B can be updated by:

$$b^*_{i,v_k} = \frac{\sum_{t=1}^{T} 1_{o_t = v_k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)} \tag{4}$$

where $1_{o_t=v_k}$ is 1 if $o_t = v_k$ and 0 otherwise.

Intuition: the update for b^*_{i,v_k} is just the frequency across the entire observation sequence of being in the state i and seeing v_k.

The new prior can be determined by

$$\pi^*_i = \gamma_i(0) \tag{5}$$

Intuition: this is the probability of being in state i before the observation sequence begins.

To update the A matrix (each a_{ij} entry), we need to figure out the probability of being in state i and *moving* to state j next. Fortunately, this calculation can be done in a straightforward way, and we can think of it as a "relative frequency calculation" of the transitions that begin at state i and go to state j compared to the transitions through state i. This "relative frequency intuition" is provided by Rabiner and Juang [6].

Define $\xi_{ij}(t)$ to be the probability of being in state i at time t and state j at time $t+1$ given $\mathbf{O}$ and parameter θ.

$$\begin{aligned}\xi_{ij}(t) &= P(X_t = i, X_{t+1} = j | \mathbf{O}, \theta) \\ &= \frac{\alpha_i(t) a_{ij} \beta_j(t+1) b_j(o_{t+1})}{\sum_{k=1}^{N} \alpha_k(t)\beta_k(t)}\end{aligned} \tag{6}$$

Finally, A can be updated by:

$$a^*_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)} \tag{7}$$

These steps are repeated until the parameters converge.

2.3 Prior Work

Here we review prior work related to *parallelization* of hidden Markov models specifically. In 2009, Chuan Liu conducted an independent project aiming to implement HMM training in parallel on CUDA. The project's parallelization forward-backward probabilities computation inspired our design. The rest of his work frames computations in terms of matrix multiplication operations. His work also utilizes "parallel reduction" for computing the sum of a vector, something we've seen in CS205 [7].

Also, we noted the approach of Li et. al in 2010, applying an earlier parallelization approach—"cut and stitch"—to the problem of hidden Markov model training. The "cut and stitch" approach parallelizes along the length of the input observation sequence (along the T-axis). The graph parallel scheme in this paper is orthogonal to the "cut and stitch" approach, and we are excited about the prospect of applying these two parallelization schemes to the problem of HMM training [8].

3 Approach and Secret Weapon

3.1 Serial Implementation

We first implemented Baum-Welch in Python with numpy, using a mix of vectorized and non-vectorized code. [1] Our initial implementation ran into serious underflow issues from the calculations of α's and β's due to multiplying probabilities together. We solve this by following the methods detailed by Zhai [9]. We normalize by scaling $\alpha_i(t)$'s such that they sum up to 1 over i's—that is over all states. So

$$\sum_{i=1}^{N} \hat{\alpha}_t(i) = 1 \tag{8}$$

and we want to define the normalizers η_k such that

$$\hat{\alpha}_t(i) = \alpha_i(t) \prod_{k=1}^{t} \frac{1}{\eta_k} \tag{9}$$

then

$$\prod_{k=1}^{t} \eta_k = \sum_{i=1}^{N} \alpha_t(i) \tag{10}$$

We can define

$$\hat{\beta}_i(t) = \beta_i(t) \prod_{k=t+1}^{T} \frac{1}{\eta_k} \tag{11}$$

then the formulas for γ's and ξ's remain the same since the normalizers cancel out.

We verify the correctness of our implementation with the library HMM function in MATLAB [10], and our calculations of α, β, and γ seemed to match, though we found some disagreement in the final results, on both corner cases and non-corner cases in our final checks. We believe that MATLAB may handle some corner cases differently, and in addition, may have a different method for computing some of the final sums necessary to update the model parameters. Regardless, we don't think we missed anything in our implementation that would cause the performance to drastically differ, and that is what we are interested in here.

3.2 Failed Parallel Attempt

We first approached the challenge by representing the Hidden Markov model computation as one over a bipartite graph. Consider the Markov chain that underlies the hidden Markov model. We created two vertices for each state in this chain, one designated the "odd vertex" and one designated the "even vertex". Call original vertex u's two vertices u_{odd} and u_{even}. For each original edge (u, v) in the hidden Markov model, we created two edges in our bipartite graph, (u_{odd}, v_{even}), (u_{even}, v_{odd}). Each vertex v, regardless of being an odd or even vertex, stores:

- Two arrays of size $\Theta(T)$, where T is the length of the observation sequence: one for each of the original state v's forward and backward probabilities. Odd vertices store forward-backward probabilities corresponding to odd time steps, and even vertices store probabilities corresponding to even time steps. Note that it is possible to save space by a constant factor in this implementation by allocating an array of roughly $T/2$ in size. In our initial implementation, we did not end up making this space optimization.

- The column of the observation matrix corresponding to its associated state in the original Markov chain, of size $\Theta(M)$. Note that in this approach, both copies of the vertex—odd and even—receive the same column of the observation matrix, though this redundant space usage does not affect our asymptotic space usage.

[1] We also used the range function instead of the xrange function in some places. Our Python implementation also uses the default data type of numpy/Python; our later parallel implementation uses a mix of doubles and floats. Our project desires to make a comparison based on scaling, so we don't consider these "constant factor" sorts of differences to be an issue.

Edges initially store the appropriate entry of the initial transition matrix A. Like the observation matrix columns, the edge a_{ij} values are duplicated, one for each version of an edge.

With this computation graph, one can now "ping pong" the computations of the forward and backward probabilities across the graph's sides. Recall the recursive case for the forward probabilities as in equation 1.

Suppose $t+1$ is an even number. Then vertex j_{even} can compute a sum based on its incoming edges, one for each original state i of the form (i_{odd}, j_{even}). Vertex i_{odd} stores the forward probability $\alpha_i(t)$, since t is an odd number. Of course, the edge joining these vertices stores the relevant transition probability a_{ij}. The relevant emission probability $b_j(O_{t+1})$ is on the vertex j_{even} too. Thus, this sum can be computed using GraphLab's `triple_apply` function, each relevant edge adding one term each to an accumulator variable on its right endpoint, j_{even}. The backward probabilities can be computed in a similar way.

Recall the equation for the next phase of the Baum-Welch training algorithm, computing γ_{it} values (as in equation 3).

This parameter is straightforward to compute in parallel, given our graph representation: every vertex can simply compute the γ_{it} for odd or even t pertaining to its original state i in the Markov chain. This computation can be done in parallel across all vertices. Note that in our implementation, the normalization scheme is such that $\sum_{j=1}^{N} \alpha_j(t)\beta_j(t) = 1$, so the division is unnecessary, which simplifies this portion of the algorithm.

Before we get to the good stuff—updating the A matrix—we need to compute one more parameter, $\xi_{i,j}(t)$ (see equation 6). Again, due to normalization, the denominator here turns out to be 1. We also noted a potential optimization that we applied in serial as well: the update of the hidden markov model parameters only relies on the sum $\sum_{t=1}^{T-1} \xi_{i,j}(t)$ for fixed (i, j), so in our implementation, we only store the sum for each (i, j).

The numerator seemed like a perfect fit for another `triple_apply` function call. An edge joins the node storing the $\alpha_i(t)$ and the node storing $\beta_j(t+1)$, since one of these t values is odd and the other is even. This edge then provides the relevant transition probability a_{ij} and the observation matrix entry can just be grabbed off the version of node j, odd or even. We realized a problem. There are, as stated before, two copies of the nodes, and two edges joining copies with different parities. The sums of the $\xi_{i,j}(t)$ values that are computed on each edge will only have access roughly half of the terms from the desired (i, j) sum, because of the duplication of nodes into odd and even versions in this approach. Communication would be required to combine the two values, and send them out to each edge for updates. We considered performing join operations on the associated edge data table, but we worried about performance of this fix and ultimately opted for what we think is a simpler approach.

3.3 Final Parallel Implementation

Our final parallel implementation uses a much simpler graph representation: the graph is simply the hidden Markov chain in question! No duplicated edges or vertices. The vertices and edges store the same data as the previous implementation, except that each vertex stores all of its forward and backward probabilities, not just half of them. By design, this approach is compact, and also distributes space for the data evenly across each of the vertices.

Computation of the forward probabilities, backward probabilities, and γ_{it} probabilities can be computed similarly to before.[2] Computation of the $\xi_{i,j}(t)$ values can be computed similar to before, except now each edge has full access to the relevant α, β values for all t, not just half of them.

[2]In our implementation, we perform the computation of the γ_{it} using vectorized arithmetic operations that GraphLab provides on the associated vertex data table.

3.4 Secret Weapon

We had a lot of secret weapons.

- **GraphLab Create SDK (C++)** [11]. Released in December 2014, this SDK "provide[s] 3rd party extensibility to GraphLab Create." After difficulty with achieving strong performance in the GraphLab programming assignment, Surat Teerapittayanon suggested that we try this SDK, and we found it powerful.

- **Dato Forum** [12]. Dato, the company behind GraphLab, was extremely helpful with our reports of bugs and other issues when working with their SDK. We even suggested an optimization for the core library of GraphLab, which they recommended we implement and submit in a pull request!

- **MATLAB** [10]. MATLAB comes with an HMM library in its Machine Learning Toolkit, and this library offers invaluable functions that allow one to see $\alpha_i(t)$, $\beta_i(t)$, and γ_{it} values for given training sequences, which was enormously invaluable in debugging. MATLAB also reports which normalizing values are used, and used a trial-and-error debugging approach to figure out what was wrong with our code at times. See the disclaimer earlier in the paper about matching the MATLAB implementation.

4 System Description

We benchmarked our code on the Odyssey cluster, supported by the FAS Division of Science, Research Computing Group at Harvard University [13].

5 Performance Evaluation

We evaluated the performance of our code, measuring runtime of our serial and graph parallel code for a variety of N, M, and T. We generated synthetic data with random (A, B, π) as the true parameters, and initialized our Baum-Welch training parameters randomly. For both the serial and parallel versions, we ran 5 update iterations of the Baum-Welch algorithm. Additionally, we also varied the number of cores available for GraphLab. R [14] was used for data analysis and plotting.

Our parallel code's runtime, compared to that of the serial version, is shown in Figure 1. We see that, while our GraphLab implementation runs slower than our serial implementation, the running time still decreases as we increase the number of processing cores.

Figures for other testing conditions are shown in the appendix, in Figures 3–6. In general, runtime for the serial code remains constant, and runtime for the parallel code decreases with more processors. In many cases, runtime increases when going from 2 to 4 cores—this may represent increased communication cost between more processors. Noise in our plots may be caused by changes in processing load on the Odyssey server.

We examined how our implementation scales as we increase the problem size, especially N. A log-log plot of running time vs. N is shown in Figure 2. Judging by the slopes on the graph, our GraphLab code scales better than the serial code for smaller N, but unfortunately not for other values of N.

Additional figures for other testing conditions display similar trends and are shown in the appendix, Figures 7–10.

6 Conclusion

Our implementation demonstrates how HMMs can be parallelized in a graph-parallel framework. We view this project is a success, because it provides a strong foundational implementation for others to parallelize this nontrivial machine learning algorithm—in a framework that's gaining traction fast. We intend to share our work with Dato, for example, since they already provide GraphLab users with a host of

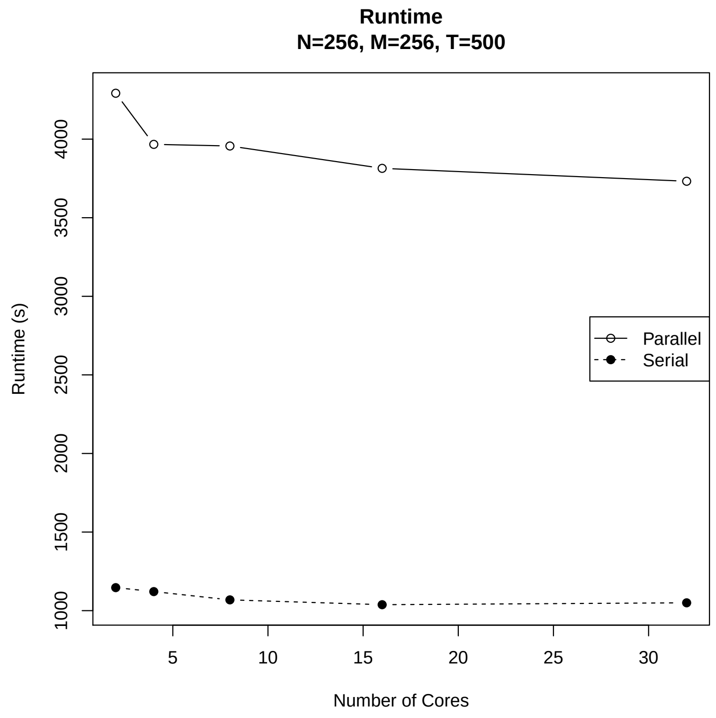


Figure 1: Runtime as a function of number of cores used.

other machine learning algorithms and preprocessing routines, but not HMM representation / training. Even though our graph-parallel approach is slower than our serial approach, we show that it achieves better scaling than the serial approach in limited certain cases. We conjecture that the difference in runtime relates to GraphLab's communication cost. We also believe that GraphLab's immutable graph data structure, `SGraph` may be slow to manipulate and may be copied often. Finally, we believe that on some `triple_apply` calls, the core code may be unnecessarily locking the vertex data, since the locking is performed even when vertex data is not mutated, but simply read. We have suggested this optimization to the Dato team and they have endorsed this suggestion as an area for future improvement in their product, encouraging us to submit a pull request to their open source core code. All put together, we are very optimistic about the work we have done and enjoyed getting our feet wet with a framework that we were previously less-than-comfortable with.

Acknowledgement

The authors like to thank the teaching staff of CS 205, as well as Jay Gu and Srikrishna Sridhar from Dato, for their tips and helpful advice.

References

[1] Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[2] Hyeon-Kyu Lee and Jin-Hyung Kim. An hmm-based threshold model approach for gesture recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 21(10):961–973, 1999.

[3] Yucheng Low, Joseph E. Gonzales, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. Graphlab: a new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Confer-*

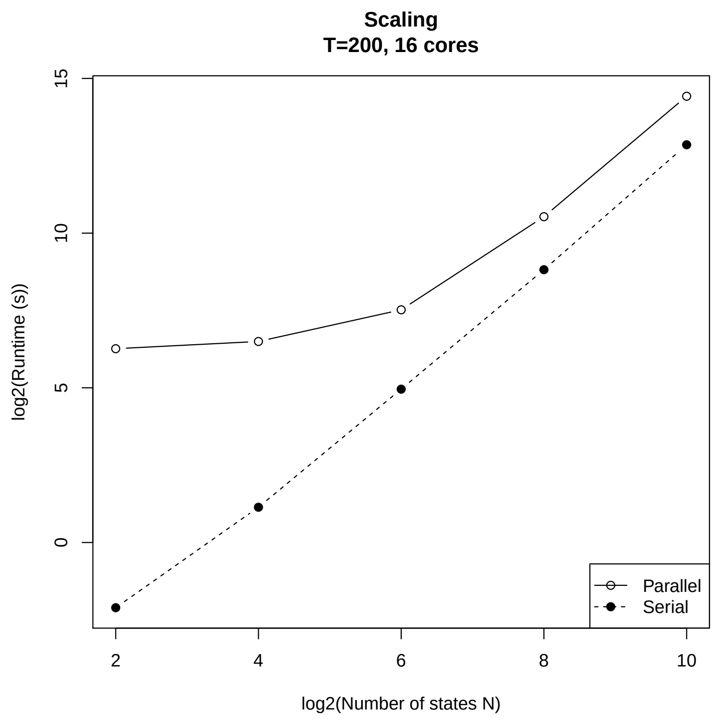


Figure 2: Log-Log plot of runtime as a function of number of states N.

ence on Uncertainty in Artificial Intelligence (UAI2010). UAI, 2010.

[4] Dato. dato.com.

[5] Wikipedia. Baum-welch algorithm— wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm#Algorithm, 2015. [Online; accessed 11-May-2015].

[6] Lawrence Rabiner and Biing-Hwang Juang. An introduction to hidden markov models, 1986.

[7] Chuan Liu. cuhmm: a cuda implementation of hidden markov model training and classification. *Johns Hopkins University*, 2009.

[8] LI Lei, FU Bin, and Christos Faloutsos. Efficient parallel learning of hidden markov chain models on smps. *IEICE transactions on information and systems*, 93(6):1330–1342, 2010.

[9] ChengXiang Zhai. A brief note on the hidden markov models (hmms). *Department of Computer Science University of Illinois at Urbana-Champaign*, 2003.

[10] *MATLAB R2015a*. The MathWorks Inc., Natick, MA, 2015.

[11] Graphlab create 1.x sdk (beta). https://dato.com/products/create/sdk/docs/index.html.

[12] Dato forum. http://forum.dato.com/.

[13] The Presidents and Fellows of Harvard University. Research computing. rc.fas.harvard.edu.

[14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

A Appendix of Figures

This section includes all of our figures for both runtime and scaling, for all of the parameters tested. Figures 3–6 show parallel and serial runtime as a function of the number of cores used, while Figures 7–10 show log-log plots of runtime as a function of N, the number of hidden states in our HMM.

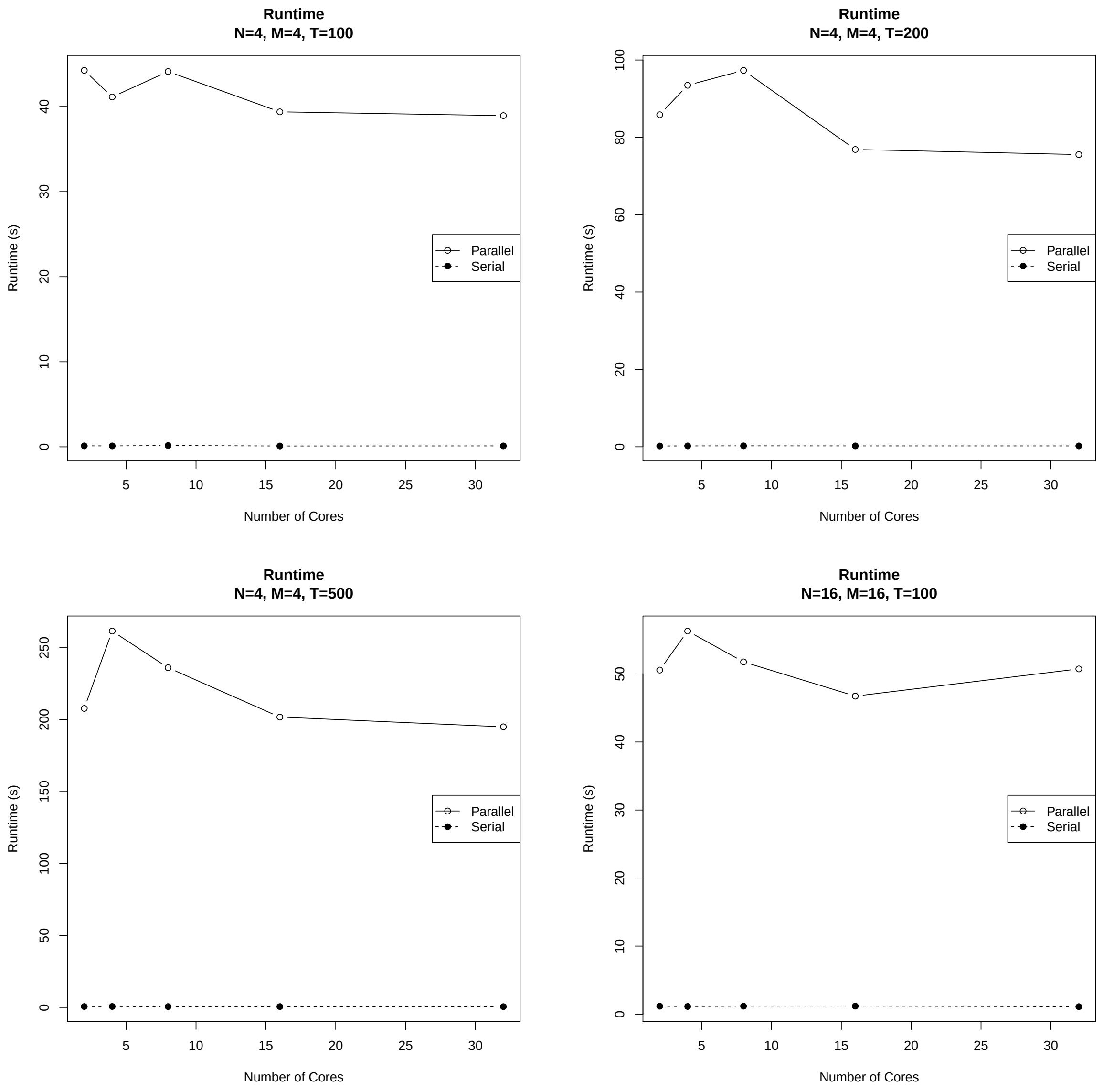


Figure 3:

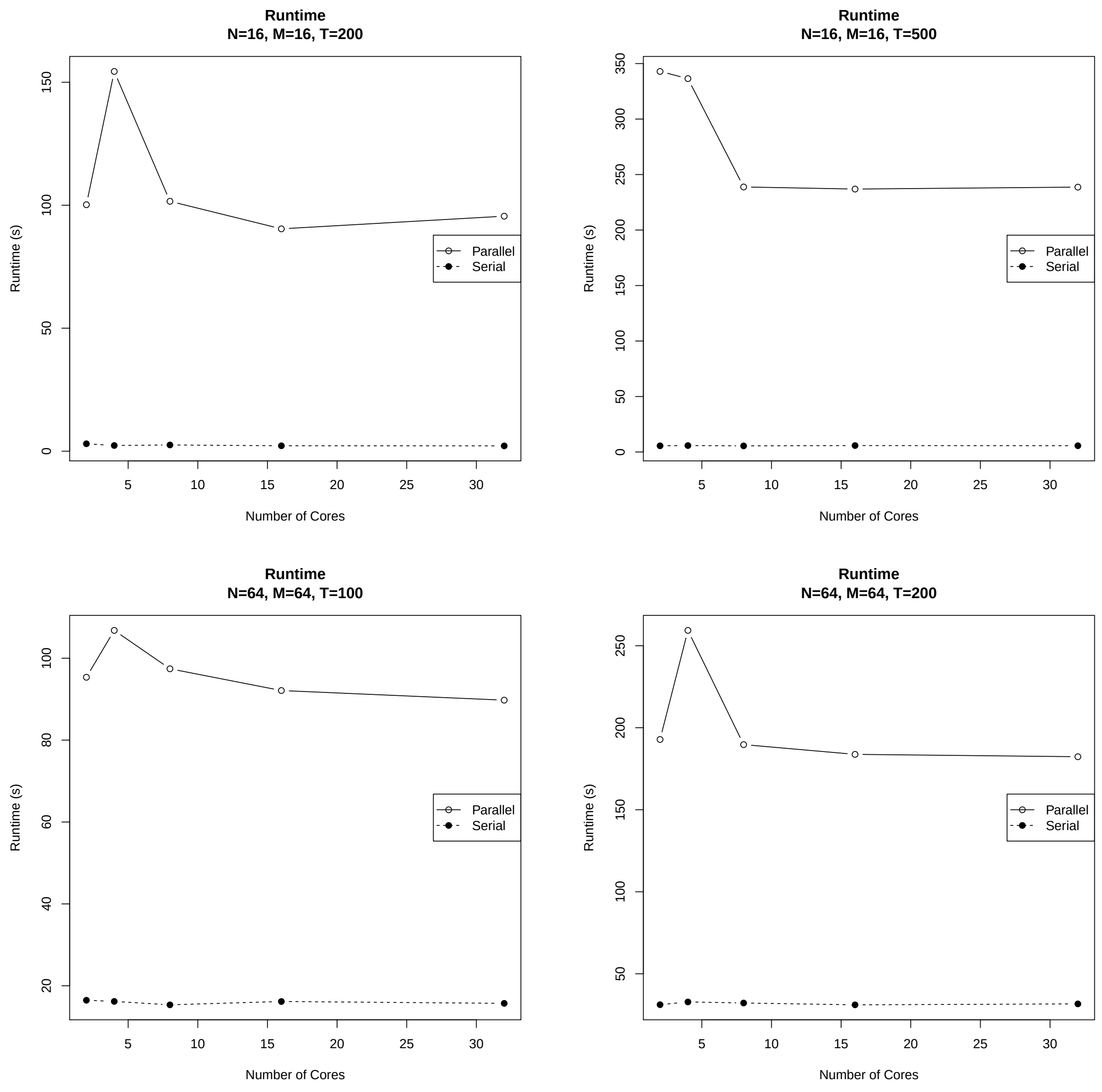
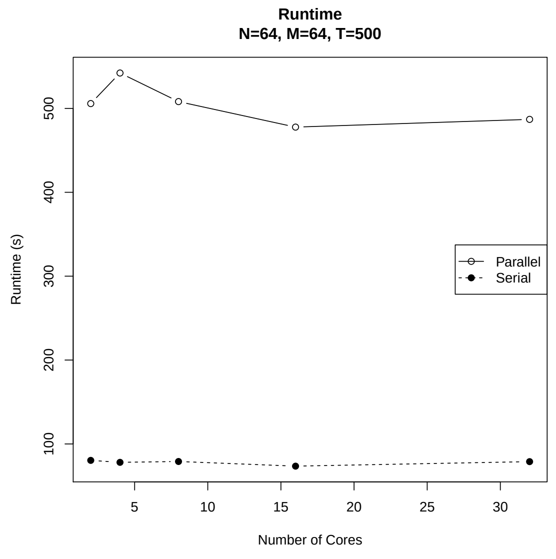
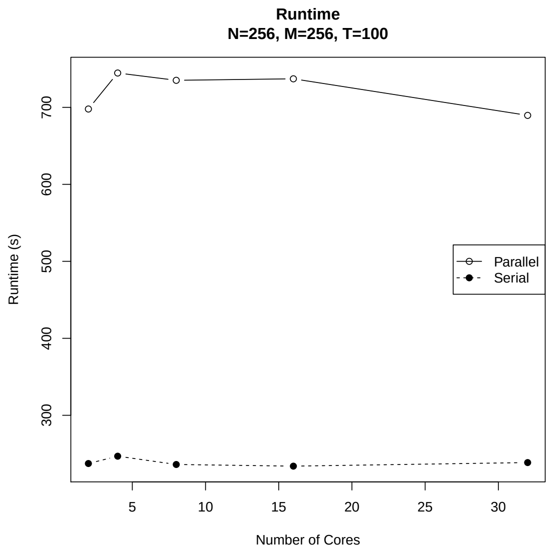
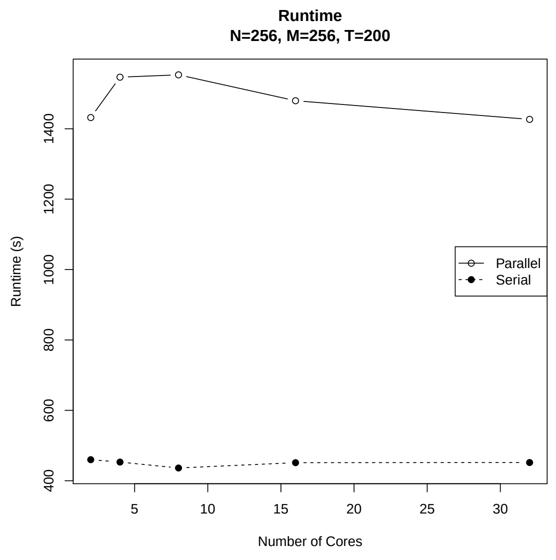

Runtime
N=16, M=16, T=200
Runtime
N=16, M=16, T=500
Runtime
N=64, M=64, T=100
Runtime
N=64, M=64, T=200
Runtime (s)
Number of Cores
Parallel
Serial

Figure 4:


Runtime
N=64, M=64, T=500
Runtime (s)
Number of Cores
Parallel
Serial


Runtime
N=256, M=256, T=100
Runtime (s)
Number of Cores
Parallel
Serial


Runtime
N=256, M=256, T=200
Runtime (s)
Number of Cores
Parallel
Serial

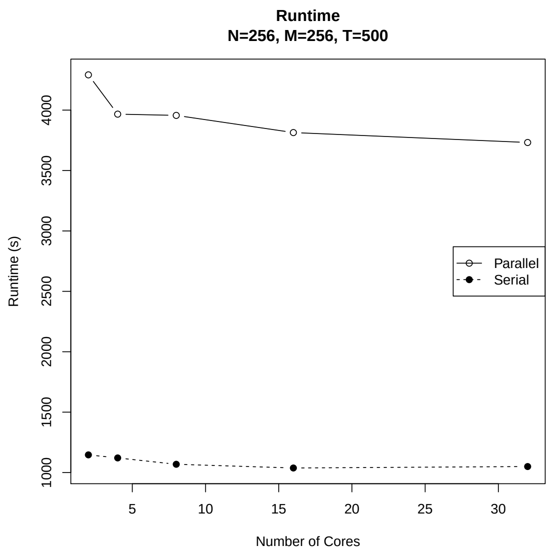
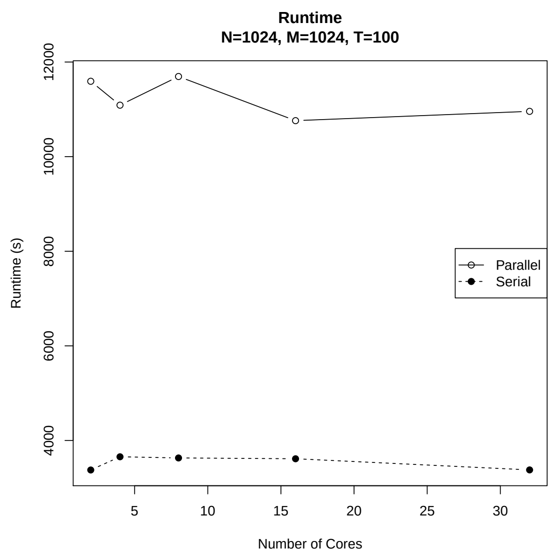

Runtime
N=256, M=256, T=500
Runtime (s)
Number of Cores
Parallel
Serial

Figure 5:


Runtime
N=1024, M=1024, T=100
Runtime (s)
Number of Cores
Parallel
Serial
12000
10000
8000
6000
4000
5
10
15
20
25
30

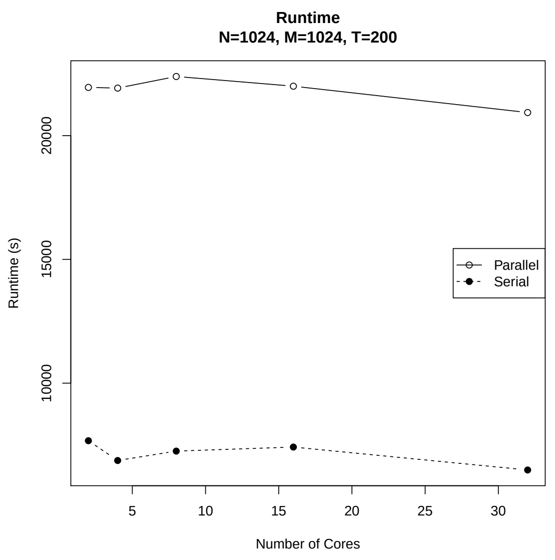

Runtime
N=1024, M=1024, T=200
Runtime (s)
Number of Cores
Parallel
Serial
20000
15000
10000
5
10
15
20
25
30

Figure 6:

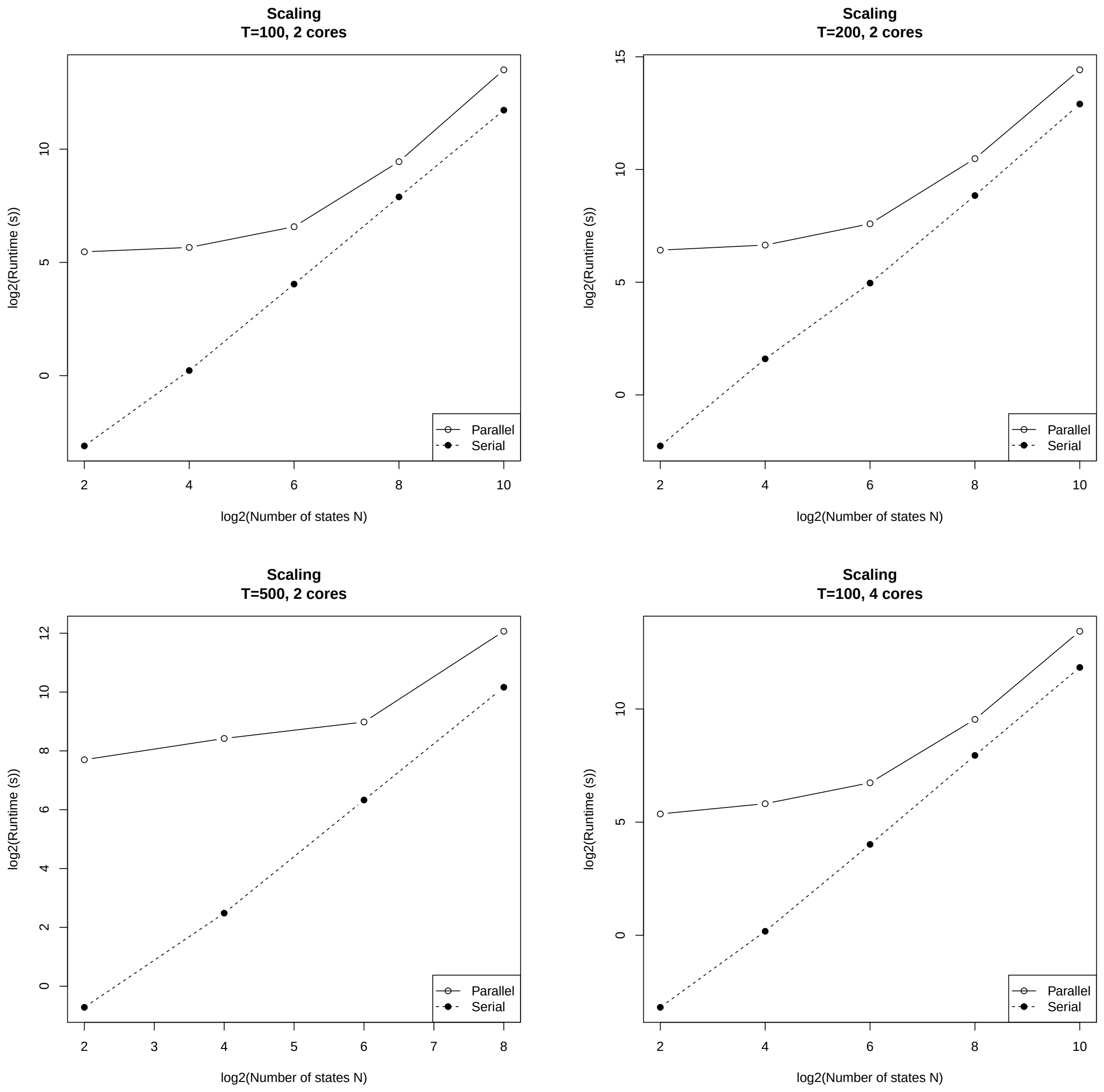

Scaling
T=100, 2 cores
Scaling
T=200, 2 cores
Scaling
T=500, 2 cores
Scaling
T=100, 4 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial

Figure 7:

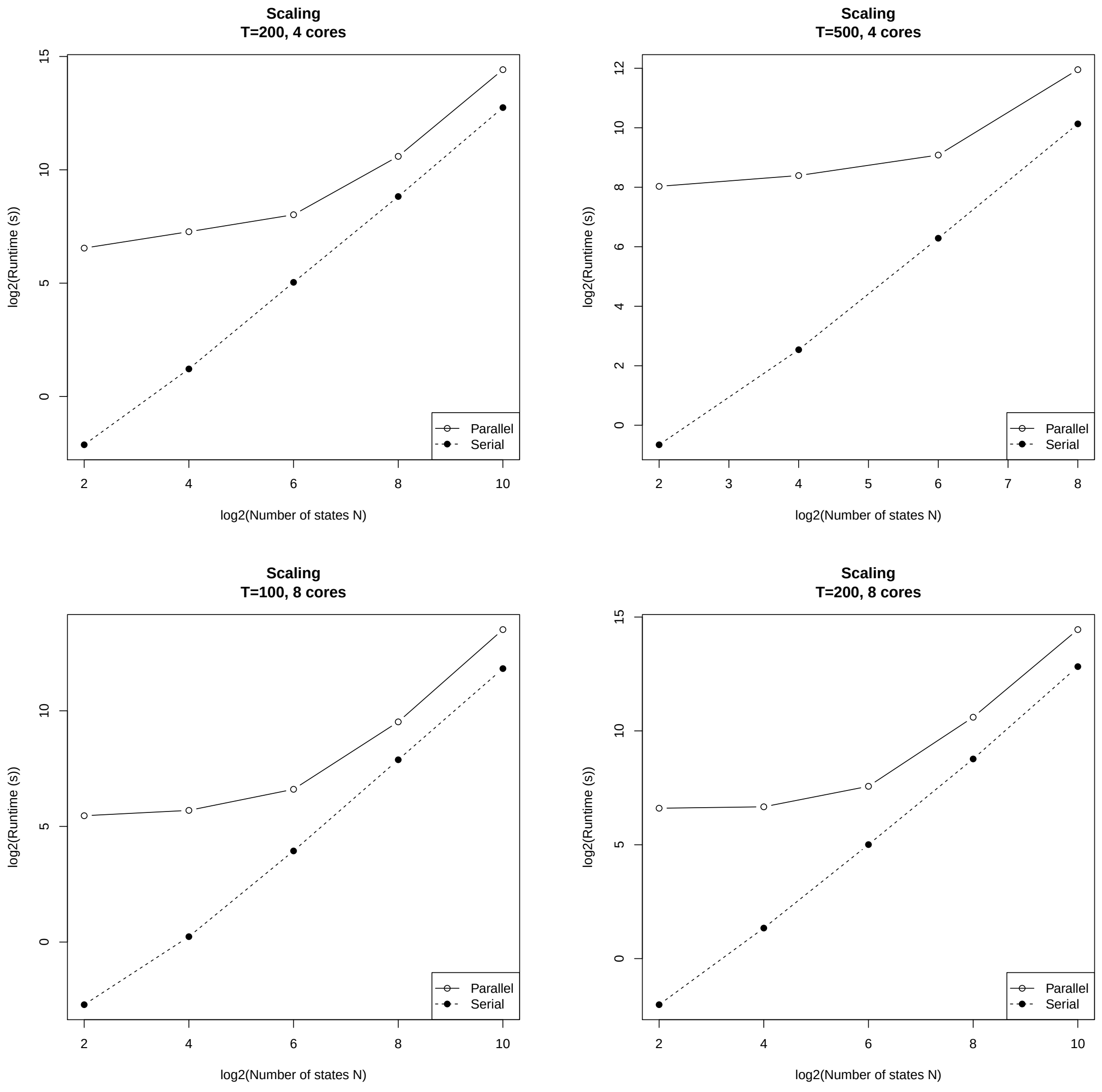

Scaling
T=200, 4 cores
T=500, 4 cores
T=100, 8 cores
T=200, 8 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial
0
5
10
15
2
4
6
8
12
3
7

Figure 8:

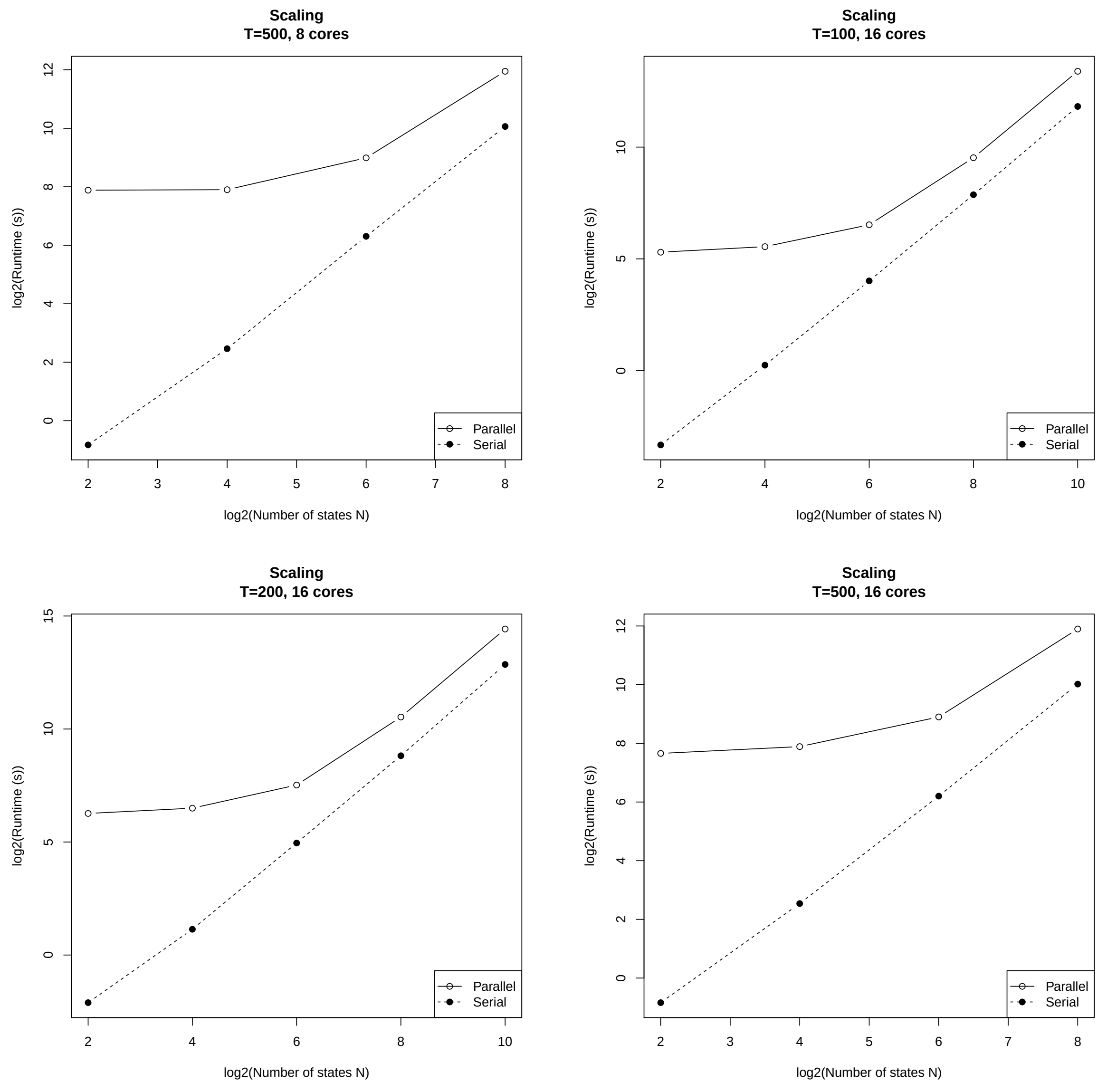

Scaling
T=500, 8 cores
Scaling
T=100, 16 cores
Scaling
T=200, 16 cores
Scaling
T=500, 16 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial

Figure 9:

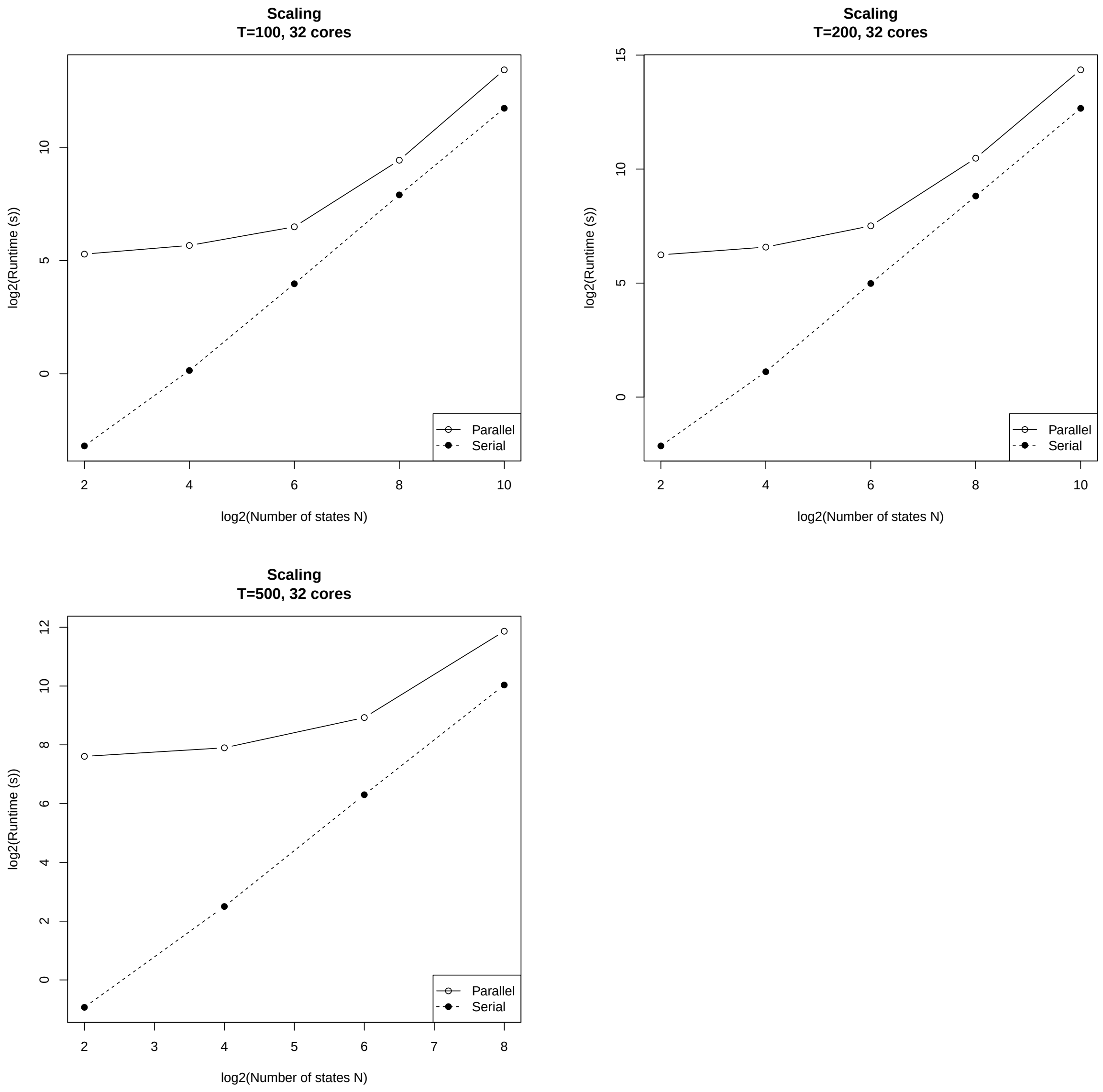

Scaling
T=100, 32 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial
Scaling
T=200, 32 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial
Scaling
T=500, 32 cores
log2(Runtime (s))
log2(Number of states N)
Parallel
Serial

Figure 10: