Maximum Likelihood Estimation of the Random-Clumped Multinomial Model as Prototype Problem for Large-Scale Statistical Computing

Andrew M. Raim[a*], Matthias K. Gobbert[a], Nagaraj K. Neerchal[a] & Jorge G. Morel[b]

[a] *Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, MD, U.S.A.*

[b] *Biometrics and Statistical Sciences Department, Procter & Gamble Company, Cincinnati, OH, U.S.A.*

Abstract

We present our application of parallel computing to the maximum likelihood estimation (MLE) of a model with large number of parameters. For many problems, MLEs must be obtained by numerical methods utilizing a computer. In some situations long computation times can become an issue as well. We consider one such problem, computing MLEs for the Random-Clumped Multinomial distribution. We compute MLEs for this model in parallel using the Toolkit for Advanced Optimization (TAO) software library. The computations are performed on a distributed-memory cluster with low latency interconnect. We study how the resource requirements change as problem sizes vary, and demonstrate that scaling the number of processes improves wall clock time significantly for larger problems. An illustrative example is also included, showing how parallel MLE computation could be useful in practice. Our experience with a direct numerical approach indicates that more substantial gains may be obtained by making use of the specific structure of the Random-Clumped model.

Key Words: parallel computing, maximum likelihood estimation, mixture distribution, multinomial

AMS Subject Classification: 65C60, 65Y05

1 Introduction

Consider a random sample of n observations $\boldsymbol{X} = (X_1, \ldots, X_n)$ drawn from a probability distribution $f(x \mid \boldsymbol{\theta})$. The vector of parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_q)$ will be considered unknown, and belongs to the space $\Theta \subseteq \mathbb{R}^q$. For a given dataset $\boldsymbol{x} = (x_1, \ldots, x_n)$, the likelihood is given by

$$L(\boldsymbol{\theta} \mid \boldsymbol{x}) = \prod_{i=1}^{n} f(x_i \mid \boldsymbol{\theta}),$$

*Corresponding author. Email: araim1@umbc.edu

and the maximum likelihood estimate (MLE) is obtained by

$$\widehat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta} \mid \boldsymbol{x}),$$

or equivalently by maximizing the log-likelihood $\log L(\boldsymbol{\theta} \mid \boldsymbol{x})$. In some situations maxima can be calculated analytically. Often however, numerical methods are required to carry out the optimization. Some commonly used numerical methods include expectation maximization (EM), Newton-Raphson, and Fisher Scoring. See [1] for a general overview.

We investigate the computation of MLEs for the Random-Clumped Multinomial distribution using a parallel architecture, as originally outlined in the technical report [2]. In particular, we make use of a publicly available software library TAO (Toolkit for Advanced Optimization) [3], which implements a number of commonly used numerical optimization methods. TAO is a special optimization library, designed for use in parallel computing environments. The objective of this work is to study the effectiveness of applying parallel optimization to the MLE problem, in terms of computing time. We make use of the *limited-memory, variable-metric* (LMVM) unconstrained optimization method in TAO. Malouf [4] has demonstrated the effectiveness of LMVM in the setting of natural language processing. Using TAO to conduct maximum entropy estimation, he shows that LMVM outperforms several other methods such as conjugate gradient. In the present work, LMVM is used to compute maximum likelihood estimates for the Random-Clumped distribution introduced by Morel and Nagaraj in [5], for which numerical optimization is necessary.

In a 2003 report describing the now-popular R package SNOW (Simple Network of Workstations), Rossini, Tierney, and Li [6] noted that parallel computing had not yet been widely adopted by statisticians. Since then, packages like SNOW and its successors have helped make parallel computing more accessible to R programmers. Many of these packages are geared toward *embarrassingly parallel* problems — those which can be easily decomposed into smaller problems that have little dependence on each other. For example SNOW provides a function called `parApply`, which evaluates a given function on each row (or column, or element) of a given matrix. Each row can be operated on independently, and the package determines how to allocate the work among available parallel processes. This is adequate for many problems in statistical computing, which involve repeating the same calculation many times using randomly generated inputs. Resampling methods such as the Bootstrap and Monte Carlo (MC) simulation fall into this class of applications. The methods presented in this paper probe deeper into the structure of the computations, and seek to improve performance within the algorithm itself. Therefore, they are best evaluated on a single complex problem rather than on problems involving repeated computations.

Our objective is to apply parallel computing to the MLE optimization problem, which does not fit the mold of embarrassingly parallel. The numerical optimization we utilize is an iterative process where each step must occur sequentially. We would like to distribute the work across many parallel processes so that the computing time can be reduced. To effectively use many processes we split the workload at each iteration, then distribute the results back to all processes to prepare for the next iteration. Therefore efficient communication is important for getting good performance, and high performance computing (HPC) is especially suitable for this application. An HPC cluster provides an array of fast processors connected by a low-latency high-throughput interconnect, and optimized communication software such as the Message Passing Interface (MPI). This environment will ensure that processes can

communicate efficiently, and that we can benefit from scaling the procedure to run on a large number of processes.

In Section 2 the Random-Clumped model is described, including an algorithm for drawing samples from the distribution. In Section 3 the approach for computing MLEs in parallel is discussed. In Section 4 simulation studies are presented. We conduct experiments to study how run times and solution quality are affected as experiment variables are changed, and verify that the correct behavior is occurring. We also study how the parallel performance is affected by changing some of the variables of the experiment. We find that changing some of the problem dimensions increases the difficulty of estimation very quickly but that the parallel performance is very good overall for a fixed problem size. In Section 5 we present a simulated scenario which is well-suited to modeling by the Random Clumped Multinomial, but inference is computationally expensive. The parallel MLE method is applied to significantly improve the performance of a likelihood ratio test. Finally, concluding remarks are given in Section 6.

2 The RCM model

First, let us consider the standard multinomial distribution, which arises in a natural way when a group of m people are asked a survey question with k possible responses. The sample space is the discrete set

$$\mathcal{T} = \left\{ \boldsymbol{t} = (t_1, \ldots, t_k) : \; t_j \in \{0, 1, \ldots, m\}, \; \sum_{j=1}^{k} t_j = m \right\},$$

where t_i denotes the number of people who gave the ith response. Let $\boldsymbol{T} = (T_1, \ldots, T_k)$ denote a random vector of counts from $\mathcal{T}$. If we assume that the participants respond to the question independently of each other, with probabilities $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$ corresponding to the possible responses, $\boldsymbol{T}$ will be distributed according to the multinomial distribution, whose density function is

$$f(\boldsymbol{t} \mid \boldsymbol{\pi}, m) = \frac{m!}{t_1!\, t_2! \cdots t_k!}\, \pi_1^{t_1}\, \pi_2^{t_2} \cdots \pi_k^{t_k}, \quad \boldsymbol{t} \in \mathcal{T}. \tag{1}$$

The parameter space is then

$$\Theta = \left\{ \boldsymbol{\pi} \in \mathbb{R}^k : \; 0 \le \pi_j \le 1, \; \sum_{j=1}^{k} \pi_j = 1, \right\},$$

with only $k-1$ distinct parameters since $\pi_k = 1 - \sum_{j=1}^{k-1} \pi_j$. If a random vector $\boldsymbol{T}$ follows this distribution, we write $\boldsymbol{T} \sim \text{Mult}(\boldsymbol{\pi}, m)$, and denote observed data as $\boldsymbol{t} = (t_1, \ldots, t_k)$. If we repeat this survey n times, each time with a group of m people, we will obtain a sample $\boldsymbol{X} = (\boldsymbol{T}_1, \ldots, \boldsymbol{T}_n)$, which can be thought of as a $k \times n$ matrix.

A mixture of ν multinomials constructed with mixing proportions $\boldsymbol{w} = (w_1, \ldots, w_\nu)$ is given by

$$f(\boldsymbol{t} \mid \boldsymbol{w}, \boldsymbol{\pi}, m) = \sum_{j=1}^{\nu} w_j\, f(\boldsymbol{t} \mid \boldsymbol{\pi}_j, m), \tag{2}$$

where $\sum_{j=1}^{\nu} w_j = 1$, $0 < w_j < 1$ for $j = 1, \ldots, \nu$, and $\boldsymbol{\pi}_j = (\pi_{j1}, \ldots, \pi_{jk})$ is the vector of probabilities corresponding to the jth component of the mixture. One motivation for considering a mixture distribution may be drawn from the point of view of classification. Suppose the participants in our multinomial response survey are drawn from one of ν different populations, and we are unable to record the population label for each subject. Of course, if the population label were available, we will end up with ν independently distributed multinomial count vectors. Since the labels are not available, the likelihood will be based on the mixture density given above in (2). This distribution has been widely used in a number of applications including text mining, linguistics, and clustering. See [7] for a detailed review. These mixture likelihoods generally cannot be maximized in closed form, as opposed to the standard multinomial, and so numerical methods are suitable for the MLE problem. Mixtures in general may not be identifiable without additional assumptions.

As the test problem for our exploration, we consider a special multinomial mixture proposed by Morel and Nagaraj [5]. Following [8], we will refer to this model as the Random-Clumped Multinomial (RCM) model. The model is also described in detail in [9]. It has more recently been referred to as the Neerchal-Morel distribution by Zhou & Lange [10], who use it to help demonstrate the minorization-maximization principle. The motivation for the RCM model can be seen in the survey scenario mentioned earlier. If the m participants interact among themselves before providing their responses, then the key "independence" assumption is violated, and the multinomial distribution does not adequately model the responses. In fact, it can be shown that such data, due to lack of independence, exhibits larger variability than the multinomial distribution. This phenomenon is commonly referred to as *overdispersion*. Morel and Nagaraj [5] provide a model for a specific type of dependence, which turns out to be a special case of the multinomial mixture distribution in (2). In subsequent work [9, 11], they show that this model has many desirable theoretical and practical properties.

The RCM model can be obtained by correlating responses within a group by a simple logic. Imagine that the group of m respondents consists of a leader who would make his/her response public. Then the remaining members may either follow the leader or make up their own mind independently of each other and the leader. Thus, the distribution of the count vector $\boldsymbol{T}$ would conform to the representation $\boldsymbol{T} = \boldsymbol{Y}N + (\boldsymbol{X} \mid N)$, where

$$\begin{aligned} N &\sim \text{Binomial}(\rho, m), \\ \boldsymbol{Y} &\sim \text{Mult}(\boldsymbol{\pi}, 1), \\ (\boldsymbol{X} \mid N) &\sim \text{Mult}(\boldsymbol{\pi}, m - N), \end{aligned}$$

such that N and $\boldsymbol{Y}$ are independent, $0 < \rho < 1$, and $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$ is a vector of category probabilities as described for the standard multinomial. It can be shown that the density for $\boldsymbol{T}$ is

$$f(\boldsymbol{t} \mid \boldsymbol{\pi}, \rho, m) = \sum_{j=1}^{k} \pi_j \, g(\boldsymbol{t} \,|\, \boldsymbol{\eta}_j, m),$$

where

$g(\boldsymbol{t} \,|\, \boldsymbol{\eta}_j, m)$ is the density of a standard multinomial,

$$\boldsymbol{\eta}_j = \begin{cases} (1-\rho)\boldsymbol{\pi} + \rho\, \boldsymbol{e}_j & \text{if } j = 1, 2, \ldots, k-1, \\ (1-\rho)\boldsymbol{\pi} & \text{if } j = k, \end{cases}$$

and where $\boldsymbol{e}_j$ is the jth column of the identity matrix, with 1 in the jth position and 0 in all other positions. We will use the notation $\boldsymbol{T} \sim \text{RCM}(\boldsymbol{\pi}, \rho, m)$ to describe the distribution of $\boldsymbol{T}$. We have noted that RCM is a special case of the mixture distribution of (2). In this mixture however, there are only k distinct parameters $\boldsymbol{\theta} = (\pi_1, \ldots, \pi_{k-1}, \rho)$. Our objective will be to compute the MLE for $\boldsymbol{\theta}$ under this model. Although we will not make use of them in this paper, theoretical results are available in [11] and [7] which help to simplify MLE computations using a Fisher Scoring approach.

Neerchal and Morel [11] describe a method for generating random samples from the RCM distribution. We include this information as a convenience to the reader. We first consider generation of samples from the standard multinomial distribution. Begin with a vector $\boldsymbol{t} = (t_1, \ldots, t_k)$ of k zeroes, and known parameters $(\pi_1, \ldots, \pi_k)$. Generate m observations from the uniform distribution $U_1, \ldots, U_m \overset{iid}{\sim} \text{U}(0, 1)$. For observation U_j, determine the category c such that

$$\pi_1 + \cdots + \pi_{c-1} < U_j \leq \pi_1 + \cdots + \pi_c$$

and add 1 to the count t_c. Repeat this process for $j = 1, \ldots, m$ to obtain $\boldsymbol{t}$.

To generate samples from the RCM distribution, begin with known parameters $(\pi_1, \ldots, \pi_k, \rho)$. Generate $m + 1$ observations from the standard multinomial distribution $\boldsymbol{S}, \boldsymbol{S}_1^0, \ldots, \boldsymbol{S}_m^0 \overset{iid}{\sim}$ Mult$(\pi_1, \ldots, \pi_k, 1)$, and m observations from the uniform distribution $U_1, \ldots, U_m \overset{iid}{\sim} \text{U}(0, 1)$. The entries of the new RCM observation are given by

$$\boldsymbol{t}_j = \boldsymbol{S}\, I(U_j \leq \rho) + \boldsymbol{S}_j^0\, I(U_j > \rho), \qquad j = 1, \ldots m,$$

where $I(\cdot)$ represents the indicator function.

Availability of this simple and intuitive algorithm of generating data is one of the many reasons for the choice of the RCM model as our test problem. It is identifiable for all values of $(\pi_1, \ldots, \pi_k, \rho)$ without requiring any additional assumptions. Furthermore, the dimension of the parameter space is proportional to the number of categories k. In a more general mixture of multinomial densities there are $q = \nu(k-1) + (\nu - 1)$ distinct parameters, including $k - 1$ category probabilities for each of the ν components, and $\nu - 1$ mixing proportions. This parameter space can blow up quickly if the number of categories or components is increased. Thus, the RCM model encompasses many numerical issues one may face in computing the MLE of a mixture model (e.g. multiple local maxima), without having to take on the full mixture. A further desirable property of RCM is that direct numerical optimization is effective for computing its MLEs. This is not the case for all mixture models, where a specialized approach like EM may be needed.

3 Computational Method

The High Performance Computing Facility (HPCF, http://www.umbc.edu/hpcf) at the University of Maryland, Baltimore County (UMBC) is an interdisciplinary, shared campus resource for scientific computing and research on parallel algorithms. The distributed-memory cluster `hpc` has 33 compute nodes, each with two dual-core AMD Opteron processors (four cores total, 1 MB of cache per core) operating at 2.6 GHz and 13 GB memory. The nodes are connected by a high performance InfiniBand network, and run 64-bit Red Hat Enterprise

Linux 5 as their operating system. We make use of the Portland Group C/C++ compiler with AMD Core Math Library (ACML), and the Open MPI 1.2.8 implementation of the Message Passing Interface (MPI) standard.

The *Toolkit for Advanced Optimization* (TAO, http://www.mcs.anl.gov/research/projects/tao) is an optimization library for both single-processor and massively-parallel architectures. It is built on top of the *Portable, Extensible Toolkit for Scientific Computation* (PETSc, http://www.mcs.anl.gov/petsc), a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. Both libraries are open source and were developed at Argonne National Laboratory. Both use MPI for handling interprocess communications. We make use of a private installation of TAO 1.9 and PETSc 2.3.3 on the HPCF cluster.

TAO and PETSc help to remove the burden of writing distributed code from the programmer. Management of distributed data structures can be left up to the libraries, allowing the programmer to focus on solving the problem at hand. Using the libraries this way however results in a loss of control, which may be detrimental to code performance. TAO provides a suite of optimization algorithms and a framework to use them; these are described in detail in the User Manual [3]. The programmer provides several key ingredients such as the objective function $h(\boldsymbol{\theta})$ to optimize, code to evaluate its gradient vector $\nabla h(\boldsymbol{\theta}) = \partial h(\boldsymbol{\theta})/\partial \boldsymbol{\theta}$, and code to evaluate its Hessian matrix $\boldsymbol{H}(\boldsymbol{\theta}) = \partial^2 h(\boldsymbol{\theta})/\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}^T$.

These three ingredients are used by the TAO algorithms to conduct a search for the optimal solution. Several algorithm choices are available for unconstrained optimization. The Nelder-Mead (NM) method is typically the worst performer, but requires only the objective function. The nonlinear conjugate gradient (CG) method and limited-memory, variable-metric (LMVM) method require both an objective and gradient function to be implemented. The Newton Line Search (NLS) method uses the objective, the gradient, and also the Hessian. For this work we have considered only LMVM because it performed well, yet does not require explicit formation of the Hessian.

We briefly describe the LMVM method. Given a current solution $\boldsymbol{\theta}^{(i)} \in \mathbb{R}^q$, LMVM consists of two main steps to find the next solution $\boldsymbol{\theta}^{(i+1)}$. First the direction $\boldsymbol{d}$ of the next step is found by solving the linear system

$$\boldsymbol{H}^{(i)} \boldsymbol{d} = -\nabla h(\boldsymbol{\theta}^{(i)}),$$

where $\boldsymbol{H}^{(i)}$ is an approximation to the Hessian, which is computed within the method. Computation of the approximation utilizes a limited amount of information coming from previous steps. After the direction is obtained, a line search is performed to compute the size of the next step τ, so that

$$h(\boldsymbol{\theta}^{(i)} + \tau \boldsymbol{d})$$

is minimized. There are several tuning parameters available for TAO's LMVM method, but we leave them at their default settings, except that the iteration limit was set to a very large number to ensure convergence.

Our implementation is carried out in the C++ programming language. The objective

function is based on the log-likelihood function from the RCM model described in Section 2,

$$h(\boldsymbol{\theta}) := -\log L(\boldsymbol{\theta} \mid \boldsymbol{X}) = -\log \left\{ \prod_{i=1}^{n} f(\boldsymbol{t}_i \mid \boldsymbol{\pi}, \rho, m) \right\}$$
$$= -\sum_{i=1}^{n} \log \left\{ \sum_{j=1}^{k} \pi_j \left[\frac{m!}{t_{i1}! \cdots t_{ik}!} \eta_{j1}^{t_{i1}} \cdots \eta_{jk}^{t_{ik}} \right] \right\}, \tag{3}$$

where $\eta_{j\ell}$'s are functions of $\boldsymbol{\pi}$ and ρ. We choose to work with the log-likelihood, which involves a summation rather than a product, because products of many probabilities may involve floating point numbers of very small magnitude. A negative sign is applied because TAO works to minimize objective functions by default.

To use an unconstrained optimization method in our problem, we must address the natural constraints in our parameter space. That is, $\boldsymbol{\theta} = (\pi_1, \ldots, \pi_k, \rho)$ are all probabilities which must lie between 0 and 1, and π_i's must sum to 1. To enforce the range constraint, we make use of the logistic cdf function $e^x/(1+e^x)$, which maps $x \in \mathbb{R}$ to the interval $(0,1)$. To enforce the summation constraint, we normalize the π_i's by scaling all components by $\sum_{i=1}^{k} \pi_i$. These two transformations are performed as the first step in evaluating the objective function.

To compute the gradient vector needed for LMVM, we use a finite difference approximation

$$\frac{\partial h(\boldsymbol{\theta})}{\partial \theta_j} = \lim_{d \to 0} \left(\frac{h(\boldsymbol{\theta} + d\boldsymbol{e}_j) - h(\boldsymbol{\theta})}{d} \right) \approx \frac{h(\boldsymbol{\theta} + \delta \boldsymbol{e}_j) - h(\boldsymbol{\theta})}{\delta}, \qquad j = 1, \ldots, q$$

with $\delta = 10^{-8}$. Notice that two objective function evaluations are needed to compute each component of the gradient.

There are several possible ways to achieve parallelization within the framework we have discussed. Observe that the log-likelihood function in (3) can be quite expensive to evaluate, because it requires iterating over every component of the mixture for each observation in the sample. Each step of LMVM requires evaluation of the gradient, and each evaluation of the gradient requires $2q$ evaluations of the objective function. Notice that the log-likelihood in (3) is a sum over n terms. One idea is to evaluate this sum over multiple parallel processes, and for large sample sizes we would expect good performance. We have chosen a different approach however, which is to compute the q components $\partial h(\boldsymbol{\theta})/\partial \theta_j$ of the gradient in parallel. Notice that these components can be computed independently. This approach limits the number of parallel processes to the dimension of the problem, but has the advantage that it generalizes to any objective function. In this scheme, all sample data must be available on all processes in order to evaluate the log-likelihood. This could also be seen as a drawback if the sample data is very large, and perhaps may not fit in the memory of a single process. This will not be an issue for the computations described in this paper, but may be a consideration in other problems.

We are now prepared to write down the parallel MLE algorithm

1. Split the indices $\{1, \ldots, q\}$ as evenly as possible among the p available processes. Denote $\mathrm{Ind}(s)$ as the set of indices assigned to process s

Note that because of the normalization step, we don't use the fact that $\pi_k = 1 - \sum_{i=1}^{k-1} \pi_i$ to infer π_k. We include all of the π_i parameters explicitly in the optimization, and therefore have $q = k+1$ total parameters under consideration.

2. Start with an initial guess $\boldsymbol{\theta}^{(0)} = (\theta_1^{(0)}, \ldots, \theta_q^{(0)})$
3. Run an LMVM iteration, which will subsequently invoke our gradient function
4. In our gradient function, each process s computes
$$\frac{\partial h(\boldsymbol{\theta}^{(0)})}{\partial \theta_j}, \qquad \forall j \in \mathrm{Ind}(s);$$
at this point the p processes work in parallel.
(Now each process has a fragment of the gradient vector in its local memory. To continue with the algorithm, we must make the entire vector available on all processes)
5. Make the entire vector available on all processes. This is accomplished in `MPI` with a single command `MPI_Allgather`
6. LMVM continues simultaneously on all processes, and a new guess $\boldsymbol{\theta}^{(1)}$ is obtained
7. This process repeats, giving $\boldsymbol{\theta}^{(0)}, \boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(g)}$, until stopping criteria are met. Finally $\boldsymbol{\theta}^{(g)}$ is returned as the MLE

For all experiments presented below, we select the total number of parameters to be evenly divisible by p. This is done for convenience and to demonstrate the ideal case of parallel performance with equal load balancing, but is not a limitation of the method itself. Some of the internals of LMVM potentially work in parallel as well (e.g., linear solves) which would further improve performance, but it is not immediately apparent if they are implemented this way. Stopping criteria are left to the TAO defaults, except that we have raised the limit on number of iterations as mentioned earlier. A few details of our implementation were not mentioned here, such as efficient computation of factorials in the likelihood. For a more thorough discussion of such details, the reader may check [2].

To summarize, we have posed the RCM MLE problem as a TAO optimization problem. TAO was not strictly necessary to implement the idea, but was attractive because it features a variety of optimization routines and other utilities that we might otherwise have had to program ourselves. Our method of choice is LMVM; although this is an iterative method, we have identified pieces (components of the gradient vector) which can be computed in parallel within an iteration. We elected to use a finite difference approximation to the gradient, but the parallelization could also use a closed-form expression if available. Our approach is appropriate for the case of RCM because the MLEs cannot be obtained in closed form, yet direct numerical optimization is effective for this model. The chosen approach did not take into consideration very large datasets, instead focusing on reduced computing time as the main goal.

It is worth making an important distinction here. Suppose our task was not just to compute the MLE on a single dataset, but to approximate (say) its sampling variance using a Bootstrap of 1000 repetitions. As mentioned earlier, the Bootstrap is an embarrassingly parallel method and each repetition can be handled independently. For this problem, rather than using our algorithm above it would be most efficient to split the 1000 repetitions evenly among the p processes and combine results at the end. This would eliminate almost all communication overhead. However, sometimes in the case of computing estimates for a single dataset the computing time may be prohibitively expensive, and that is the case we address with the algorithm above. Therefore, it is important to step back and evaluate the overall computing task before deciding how parallelization should be handled.

4 Computational Experiments and Results

We design a series of experiments to verify that the optimization is working correctly ("consistency experiments"), and to study parallel performance as we vary problem sizes and parallelism ("scalability experiments"). Our experiments consist of the variables

- sample size n,
- cluster size m,
- number of multinomial categories k (total parameters: $q = k + 1$),
- number of repetitions r,
- number of MPI processes p.

To determine the true values for RCM parameters in an experiment, we generate a symmetric vector $\boldsymbol{v} = (1, 2, 3, \ldots, 3, 2, 1)$ containing $k \in \{1, 2, 3, \ldots\}$ elements. We let $\boldsymbol{\pi} = \boldsymbol{v}/\sum_i v_i$, and let $\rho = 1/4$ so that the true parameters of the experiment are $\boldsymbol{\theta} = (\boldsymbol{\pi}, \rho)$. This provides a quick but deterministic construction of $\boldsymbol{\theta}$ for any valid choice of k. We generate a random sample of n observations $\boldsymbol{X} = (\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n)$ from the RCM distribution with these parameters, as described in Section 2. The objective function $h(\boldsymbol{\theta})$ given in (3) is constructed with this sample data. The TAO framework is then invoked with an initial solution $\boldsymbol{\theta}^{(0)} = (\pi_1 = 1/k, \pi_2 = 1/k, \ldots, \pi_k = 1/k, \rho = 1/2)$. The selected optimization routine (LMVM) runs until stopping criteria are reached. If the optimization is successful, a maximum likelihood estimate $\widehat{\boldsymbol{\theta}}_{\text{MLE}}$ is obtained. Using the parameters $\boldsymbol{\theta}$, the data generation and estimation phases are repeated r times yielding the independent and identically distributed RCM samples $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_r$, and the estimates $\widehat{\boldsymbol{\theta}}^{(1)}_{\text{MLE}}(\boldsymbol{X}_1), \ldots, \widehat{\boldsymbol{\theta}}^{(r)}_{\text{MLE}}(\boldsymbol{X}_r)$ which are also independent and identically distributed.

Morel and Nagaraj verify in [5] that the MLE is consistent and asymptotically normal for RCM. Therefore, as a measurement of solution quality we consider

$$\text{SSE}(\boldsymbol{\theta}, \bar{\boldsymbol{\theta}}_{\text{MLE}}) := ||\boldsymbol{\theta} - \bar{\boldsymbol{\theta}}_{\text{MLE}}||^2 = \sum_{j=1}^{q} \left(\theta_j - \bar{\theta}_{\text{MLE},j}\right)^2, \tag{4}$$

where $\bar{\boldsymbol{\theta}}_{\text{MLE}} := \frac{1}{r}\sum_{i=1}^{r} \widehat{\boldsymbol{\theta}}^{(i)}_{\text{MLE}}$ is the empirical mean of the estimates from the r realizations. SSE is then the squared distance between $\bar{\boldsymbol{\theta}}_{\text{MLE}}$ and $\boldsymbol{\theta}$, and so we expect it to become small as more information about $\boldsymbol{\theta}$ through the data becomes available. We will be able to observe this distance as the experimental variables are adjusted, and see the associated costs in terms of computing time and memory. While SSE approaching zero does not guarantee that we are computing the global maxima of the likelihood function, it does provide some evidence that the algorithm is converging to a consistent solution of the MLE problem.

Technically, the sample generation process was conducted outside of the experiments. A sample was generated for each distinct setting of (n, k, m), for the maximum number of repetitions which were needed at that setting, and stored in a file. During the "sample generation" phase in an experiment, the appropriate sample file is identified and loaded. This process ensures that any two experiments using the same variables (n, k, m, r) will use exactly the same data, and thus their results should be comparable. We record the total walltime as the number of seconds to compute the r estimates. This time includes optimization, as well as a reset of the TAO framework to its initial state between iterations. Walltime does not include calculation of sum square error, initialization of the TAO framework at the beginning

of the program, loading of sample data, or deinitialization at the end of the program. We also record the amount of global memory in kB used among all processes, starting after the time TAO is initialized until the r estimates are computed. Memory usage is computed by taking the difference between starting and ending measurements. We record the total memory usage including swap space (denoted as `VIRT` in the Linux `top` program). The memory usage we report shows our application's usage, but does not reflect the overhead of starting TAO or MPI, or the overall memory requirement of the program. We have opted to show the global memory usage (rather than per node for example) to give an impression of the magnitude of the problem across all processes.

All experiments are distributed in the same way across the HPCF cluster. For $p \leq 4$, a single compute node is used (each process will then run on its own dedicated core). For $p > 4$, we only consider p as a multiple of 4, and choose the number of nodes as $p/4$ so that all four cores are utilized on each machine. Gobbert [12] demonstrates that use of multiple cores per node is an effective strategy for distributing workloads on the HPCF cluster. We ensure that all nodes used for any experiment are reserved exclusively for us by the scheduler.

4.1 Consistency Check for MLE

Table 1 displays a summary of results, altering each of the experiment variables separately. The column "memory" shows the global memory (in kB) used across all processes, as described earlier. The column "sse" shows the quantity defined in (4). The column "iterations" represents the total number of LMVM iterations over the entire experiment (the same number will be reported on each process).

As we might suspect, increasing the sample size n causes a linear increase in run time but also causes $\bar{\boldsymbol{\theta}}_{\text{MLE}}$ to approach $\boldsymbol{\theta}$. Increasing the cluster size m also appears to have the effect of increasing run times. Evaluation of the objective function does not depend on the size of m, except for the factorial calculations in the log-likelihood. We have implemented these calculations using the optimized `lgamma_r` function in C rather than the naive recursive formula. However, it still appears that the performance of this function depends on the magnitude of m. The number of LMVM iterations appears to be increasing for larger m, with some variation depending on randomly selected dataset. Increasing m also causes $\bar{\boldsymbol{\theta}}_{\text{MLE}}$ to approach $\boldsymbol{\theta}$. This makes sense intuitively if we consider the survey example from Section 2; more participants will provide more information about the population's opinion.

Increasing the number of categories k results in a greater-than-linear increase in run time. It seems intuitive that the optimization will become more difficult as the objective function's domain increases in dimension. This can also be seen in the number of iterations, which tends to increase with k, but with some variation depending on the dataset. There appears to be no definite trend in SSE as k varies.

Increasing the number of repetitions r causes an unsurprising linear increase in run time. Also as we might guess, more repetitions of the estimation process results in $\bar{\boldsymbol{\theta}}_{\text{MLE}}$ approaching $\boldsymbol{\theta}$. The effect on run time of scaling the number of processes p is encouraging. The run times approximately halve as the number of processes double. There does not appear to be a significant impact on the solution quality or the number of iterations as p varies, which is expected because the computations should be similar, perhaps with small numerical differences.

Table 1: Results for the consistency experiments; in each of the five sections of the table, one of the experiment variables is changed while the others are held fixed.

(a) Experiments varying n

m	k	n	r	p	walltime	memory	sse	iterations
32	7	32	16	1	3.98	264	7.656e-05	362
32	7	64	16	1	7.76	264	3.509e-05	343
32	7	128	16	1	15.90	264	1.266e-05	354
32	7	256	16	1	33.36	300	1.091e-05	369
32	7	512	16	1	67.88	456	8.930e-06	376
32	7	1024	16	1	133.64	636	5.142e-06	371
32	7	2048	16	1	269.74	1216	9.101e-07	379
32	7	4096	16	1	638.07	1820	9.616e-07	446

(b) Experiments varying m

m	k	n	r	p	walltime	memory	sse	iterations
1	31	128	16	1	313.03	276	1.393e-01	175
2	31	128	16	1	247.78	276	6.273e-02	138
4	31	128	16	1	228.69	276	6.265e-02	124
8	31	128	16	1	240.43	276	6.258e-02	130
16	31	128	16	1	273.88	276	6.255e-02	143
32	31	128	16	1	354.66	276	2.364e-02	177
64	31	128	16	1	446.87	276	1.313e-05	201
128	31	128	16	1	493.54	276	4.988e-06	204
256	31	128	16	1	657.73	276	3.044e-06	262
512	31	128	16	1	647.89	276	1.184e-06	253

(c) Experiments varying k

m	k	n	r	p	walltime	memory	sse	iterations
256	3	64	16	1	0.44	264	1.139e-06	124
256	7	64	16	1	3.77	264	4.469e-06	165
256	15	64	16	1	36.89	276	4.806e-06	215
256	31	64	16	1	313.98	284	3.416e-06	250
256	63	64	16	1	2261.36	264	3.246e-06	244

(d) Experiments varying r

m	k	n	r	p	walltime	memory	sse	iterations
32	7	128	1	1	1.01	264	5.183e-04	23
32	7	128	2	1	1.97	264	4.083e-04	46
32	7	128	4	1	3.98	264	1.262e-04	90
32	7	128	8	1	7.99	264	7.135e-05	179
32	7	128	16	1	15.63	264	1.266e-05	354
32	7	128	32	1	31.53	264	1.169e-05	711
32	7	128	64	1	63.71	264	1.349e-05	1428
32	7	128	128	1	129.83	268	6.337e-06	2881

(e) Experiments varying p

m	k	n	r	p	walltime	memory	sse	iterations
256	31	256	16	1	1338.90	516	8.649e-07	267
256	31	256	16	2	681.90	524	8.649e-07	267
256	31	256	16	4	351.40	468	8.649e-07	267
256	31	256	16	8	188.40	1036	8.650e-07	267
256	31	256	16	16	107.00	1244	8.648e-07	267
256	31	256	16	32	68.02	4296	8.648e-07	267

4.2 Performance Experiments

We consider the parallel performance of the RCM estimation problem when varying sample size n, cluster size m, and number of categories k. These variables are altered along with the number of processes p. The number of repetitions r will not be considered here, because we expect a single repetition to be representative of performance. We examine walltime as well as the metrics speedup and efficiency, which are conventionally given in parallel performance studies. Let $c \in \{n, m, k\}$ be the experiment parameter under observation. Define $T_p(c)$ as the walltime in seconds to compute a problem of size c using p processes. The *speedup* is defined as $S_p(c) = T_1(c)/T_p(c)$, where $S_p(c)$ close to p suggests ideal parallel performance. The *efficiency* is defined as $E_p(c) = S_p(c)/p$, where $E_p(c)$ close to 1 suggests ideal parallel performance. When c is held constant and the number of processes p varies, the same exact input data is used. This helps to simplify comparisons between different settings of p.

Table 2 and Figure 1 show the results of the experiments varying n. We can see that for a fixed n, doubling the number of processes p shows a strong halving effect of the walltime. The effect begins to weaken when $p = 64$, and this weakening happens a bit earlier for the smallest experiment $n = 16$. The speedup and efficiency plots emphasize that the scaling of $n = 16$ is worse than the larger values of n, which can be explained by the diminishing amount of computational work to be performed. For the cases when $n > 16$, the scaling is almost identical. Table 3 and Figure 2 show the results of the experiments varying m. Again we see the definite halving effect in walltime as p is doubled, which starts to weaken around $p = 64$. In the speedup and efficiency plots, we can see that the selected settings of m show almost exactly the same scaling pattern.

Table 4 and Figure 3 show the results of the experiments varying k. Recall in our parallelization scheme that the p processes divide the work of computing the $k + 1$ entries of the gradient vector. When $p \geq k$ some processes will be left with no useful work, so these results have been omitted. We notice that for small k, the run time is too quick to justify parallelization. As k increases, run time drastically increases. For a fixed large k such as $k = 127$, doubling the number of processes p shows a strong halving effect of the walltime, which weakens as p approaches $k + 1$. This is also reflected in the speedup and efficiency plots. The most notable observation is the decrease in run time for $k = 127$, from about 41.9 minutes serially, to about 30 seconds when using all 128 processes. Similar results are obtained in the varying n and varying m experiments, where k is fixed at 127. Thus for large enough problem sizes, scaling the number of processes drastically reduces the walltime needed to compute the MLE.

The results of multiple process runs were compared to corresponding serial runs to ensure that the parallel results were correct. These results are not presented here, in the interest of brevity.

Table 2: Walltime, speedup, and efficiency varying n, for $k = 127$, $m = 64$, $r = 1$. Tests were performed with 4 processes per node, except for $p = 1$ which uses 1 process per node, and $p = 2$ which uses 2 processes per node

(a) Wall clock time in seconds

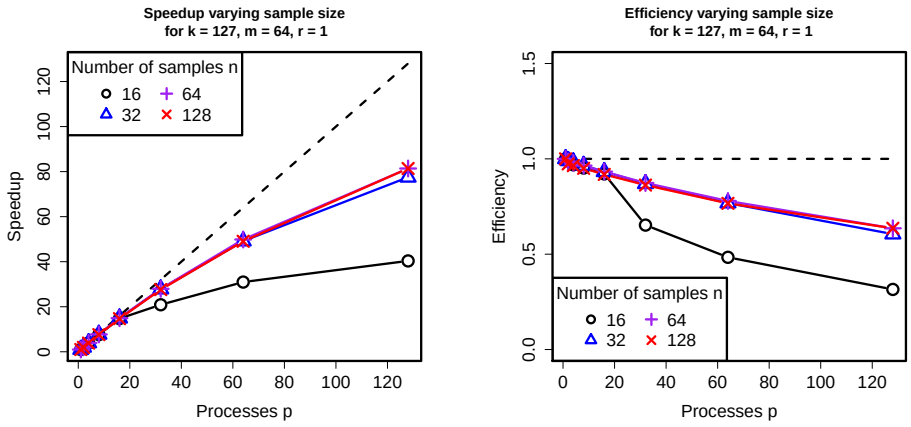
n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	265.37	133.73	68.47	34.84	18.01	12.71	8.57	6.58
32	447.10	225.88	113.71	57.91	29.96	16.07	9.09	5.77
64	1067.33	541.35	272.61	138.33	71.51	38.19	21.40	13.12
128	1752.26	900.03	454.20	230.46	119.25	63.49	35.72	21.53

(b) Observed speedup S_p

n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	1.00	1.98	3.88	7.62	14.74	20.88	30.96	40.35
32	1.00	1.98	3.93	7.72	14.92	27.83	49.16	77.49
64	1.00	1.97	3.92	7.72	14.93	27.95	49.86	81.38
128	1.00	1.95	3.86	7.60	14.69	27.60	49.06	81.39

(c) Observed efficiency E_p

n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	1.00	0.99	0.97	0.95	0.92	0.65	0.48	0.32
32	1.00	0.99	0.98	0.97	0.93	0.87	0.77	0.61
64	1.00	0.99	0.98	0.96	0.93	0.87	0.78	0.64
128	1.00	0.97	0.96	0.95	0.92	0.86	0.77	0.64



(a) Observed speedup S_p (b) Observed efficiency E_p

Figure 1: Scalability experiments — Plots showing scalability as n varies.

Table 3: Walltime, speedup, and efficiency varying m, for $n = 128$, $k = 127$, $r = 1$. Tests were performed with 4 processes per node, except for $p = 1$ which uses 1 process per node, and $p = 2$ which uses 2 processes per node

(a) Wall clock time in seconds

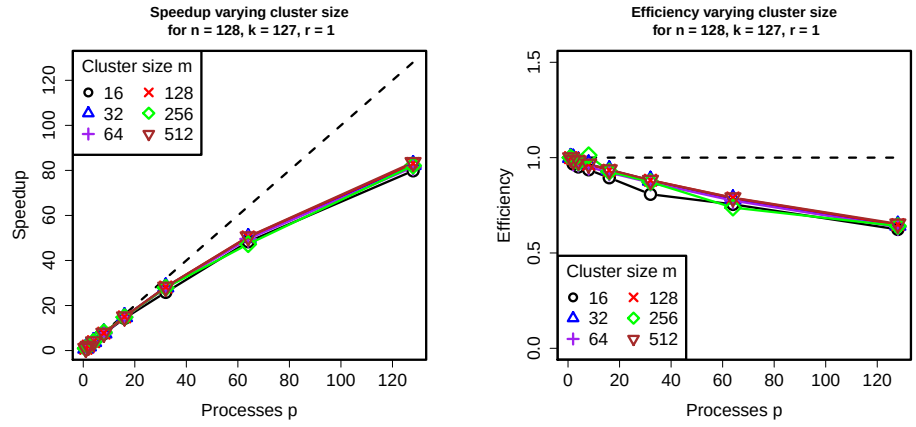
m	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	1441.94	746.13	378.99	192.51	100.74	55.78	29.84	18.06
32	1732.93	869.53	441.57	224.29	115.81	61.48	34.60	21.00
64	1758.97	902.74	453.99	230.94	119.31	63.28	35.48	21.60
128	2018.49	1014.06	513.70	261.32	135.08	71.57	40.00	24.30
256	2486.03	1257.50	637.50	306.99	167.60	89.08	52.63	30.28
512	3208.32	1625.91	815.45	414.88	214.05	113.73	63.46	38.40

(b) Observed speedup S_p

m	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	1.00	1.93	3.80	7.49	14.31	25.85	48.32	79.82
32	1.00	1.99	3.92	7.73	14.96	28.19	50.09	82.51
64	1.00	1.95	3.87	7.62	14.74	27.80	49.58	81.42
128	1.00	1.99	3.93	7.72	14.94	28.20	50.46	83.06
256	1.00	1.98	3.90	8.10	14.83	27.91	47.24	82.11
512	1.00	1.97	3.93	7.73	14.99	28.21	50.56	83.55

(c) Observed efficiency E_p

m	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
16	1.00	0.97	0.95	0.94	0.89	0.81	0.75	0.62
32	1.00	1.00	0.98	0.97	0.94	0.88	0.78	0.64
64	1.00	0.97	0.97	0.95	0.92	0.87	0.77	0.64
128	1.00	1.00	0.98	0.97	0.93	0.88	0.79	0.65
256	1.00	0.99	0.97	1.01	0.93	0.87	0.74	0.64
512	1.00	0.99	0.98	0.97	0.94	0.88	0.79	0.65



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 2: Scalability experiments — Plots showing scalability as m varies.

Table 4: Walltime, speedup, and efficiency varying k, for $n = 128$, $m = 256$, $r = 1$. Tests were performed with 4 processes per node, except for $p = 1$ which uses 1 process per node, and $p = 2$ which uses 2 processes per node

(a) Wall clock time in seconds

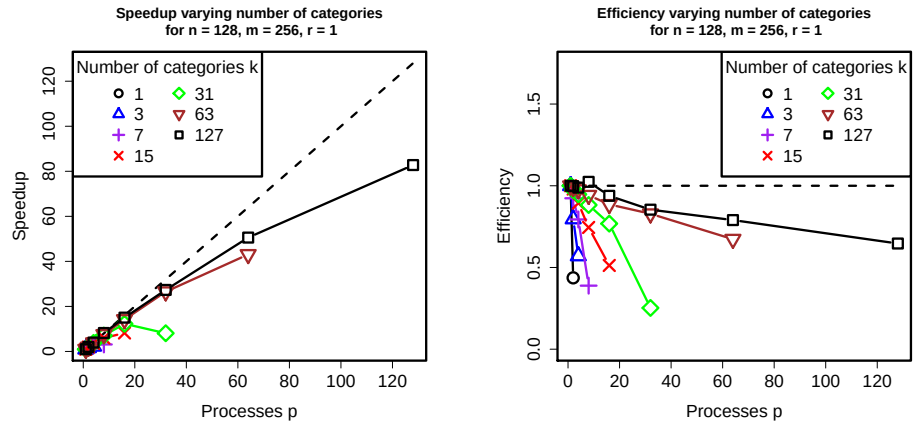
k	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
1	0.004	0.005	—	—	—	—	—	—
3	0.058	0.036	0.025	—	—	—	—	—
7	0.471	0.255	0.148	0.151	—	—	—	—
15	4.913	2.507	1.375	0.824	0.599	—	—	—
31	41.724	21.414	11.010	5.912	3.394	5.165	—	—
63	390.403	197.000	99.962	51.808	27.544	14.721	9.063	—
127	2513.446	1259.716	635.585	306.806	167.395	92.008	49.710	30.367

(b) Observed speedup S_p

k	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
1	1.00	0.87	—	—	—	—	—	—
3	1.00	1.59	2.28	—	—	—	—	—
7	1.00	1.85	3.18	3.11	—	—	—	—
15	1.00	1.96	3.57	5.96	8.20	—	—	—
31	1.00	1.95	3.79	7.06	12.29	8.08	—	—
63	1.00	1.98	3.91	7.54	14.17	26.52	43.08	—
127	1.00	2.00	3.95	8.19	15.02	27.32	50.56	82.77

(c) Observed efficiency E_p

k	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
1	1.00	0.44	—	—	—	—	—	—
3	1.00	0.80	0.57	—	—	—	—	—
7	1.00	0.92	0.79	0.39	—	—	—	—
15	1.00	0.98	0.89	0.74	0.51	—	—	—
31	1.00	0.97	0.95	0.88	0.77	0.25	—	—
63	1.00	0.99	0.98	0.94	0.89	0.83	0.67	—
127	1.00	1.00	0.99	1.02	0.94	0.85	0.79	0.65



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 3: Scalability experiments — Plots showing scalability as k varies.

5 An LRT application

Our performance studies have demonstrated the effectiveness of parallel computing for the RCM MLE problem when many parameters need to be estimated. But where might we encounter such problems in a data analysis? To answer this question, we will next consider hypothesis testing in a fixed effects model embedded into RCM. Even a fairly simple problem in this framework can be computationally expensive, if there are a large number of multinomial categories and/or covariates. To create an effective demonstration, we will consider a scenario which is ideally suited to RCM. It will feature both a large number of categories and a simple fixed effects model. We will generate data for this scenario, and show that computation of the likelihood ratio test (LRT) is one instance where practical use can be made of our parallel MLE idea.

Suppose there are n guidance counselors who advise high school students in selecting a college from k possibilities. For simplicity, suppose m students are assigned to each counselor, and no student is assigned to more than one counselor. A student visits their counselor zero or more times for advice until they have chosen a college, and may or may not be influenced by their counselor. Let $\boldsymbol{x} = (x_1, \ldots, x_n)$, where x_i is the total number of visits to counselor i by their students. Intuitively, we might expect that a more heavily utilized counselor will have a greater influence on their students. One can imagine each student choosing between the counselor's recommendation and his/her own personal choice, as in the generation of RCM described in Section 2. This scenario is ideally modeled by RCM with parameter ρ capturing the degree of influence of the counselor.

Let $\boldsymbol{T}_i = (T_{i1}, \ldots, T_{ik})$ denote the vector of counts for counselor i, for each of the k possible colleges. We can also suppose that the first $k-1$ categories represent specific college choices, and the kth category represents a catch-all for all other possibilities, such as attending an unlisted college or not attending any college at all. We will suppose that $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_n$ are independent, and

$$\boldsymbol{T}_i \sim \text{RCM}(\boldsymbol{\pi}, \rho_i, m), \qquad \log\left(\frac{\rho_i}{1-\rho_i}\right) = \alpha + \beta x_i. \tag{5}$$

Recall that ρ_i is the probability of "following the leader" from Section 2. In this scenario, following the leader means choosing the preferred college of the counselor. In 5 we have expressed the log-odds of ρ_i by a linear function, where α is the common baseline effect of a counselor's influence on students, and β is a common slope which incorporates how heavily students have utilized their counselor. Here $\boldsymbol{\pi}$ is constant across all counselors, and so aside from counselor influence, the probability distribution of choosing among the k colleges is the same for all students. For this problem, the unknown parameter $\boldsymbol{\theta}$ is contained in the space

$$\Theta = \left\{ (\pi_1, \ldots, \pi_k, \alpha, \beta) \in \mathbb{R}^q : \ 0 \le \pi_j \le 1, \ \sum_{j=1}^{k} \pi_j = 1 \right\}, \quad q = k + 2.$$

We will consider a testing problem for the significance of the slope,

$$H_0 : \beta = 0 \qquad \text{vs.} \qquad H_1 : \beta \neq 0.$$

Two MLE computations are needed for the LRT: the unrestricted MLE $\widehat{\boldsymbol{\theta}}$, and the MLE $\widehat{\boldsymbol{\theta}}_0$

under the restriction H_0. The LRT statistic can then be computed as

$$-2\log\Lambda = -2\log\frac{L(\widehat{\boldsymbol{\theta}}_0)}{L(\widehat{\boldsymbol{\theta}})} = -2\left\{\log L(\widehat{\boldsymbol{\theta}}_0) - \log L(\widehat{\boldsymbol{\theta}})\right\}, \tag{6}$$

where the likelihood function L is given by

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{n} f(\boldsymbol{t}_i \mid \boldsymbol{\pi}, \rho_i(\alpha,\beta), m) \tag{7}$$

and f is the density of RCM.

As before, solving the likelihood equation in closed form is not practical, so we turn to numerical computation of the MLE. This can be accomplished in parallel by simply applying the method from Section 3 and embedding the new likelihood (7) into (3). For this application we use the initial guess $\boldsymbol{\theta}^{(0)} = (\boldsymbol{\pi}^{(0)}, \alpha^{(0)}, \beta^{(0)})$, where $\boldsymbol{\pi}^{(0)} = (1/k, \ldots, 1/k)$, $\alpha^{(0)} = 0$, and $\beta^{(0)} = 0$.

To generate data from this scenario, we first select a number of categories k and sample size n, and let $m = 100$ students per counselor. The category probabilities are generated by drawing a random sample $U_1, \ldots, U_k$ from $U(0,1)$ and then letting $\pi_j = U_j / \sum_i U_i$. To generate the covariate $\boldsymbol{x}$, we suppose $x_{i1}, \ldots, x_{im} \overset{iid}{\sim} \text{Geometric}(\phi)$, where x_{ij} represents the number of visits of the jth student of counselor i until a college decision is made. We choose $\phi = 0.9$ so that the expected number of visits per student $\text{E}(x_{ij}) = (1-\phi)/\phi = 1/9$ is small. The total number of visits x_i to counselor i can then be drawn from $\text{NegBin}(m, \phi)$. We let $\alpha = -5$ and $\beta = 0.3$, so that

$$\log\left(\frac{\rho_i}{1-\rho_i}\right) = -5 + 0.3\,x_i \quad \Longleftrightarrow \quad \frac{\rho_i}{1-\rho_i} = e^{-5}(e^{0.3})^{x_i},$$

and so the odds of "following the leader" will be multiplied by $e^{0.3} \approx 1.35$ for each visit to the counselor. In this scenario, utilization of the counselor has a fairly strong effect on their influence over college choices, but students do not tend to make much use of their counselor. Now all of the parameters and covariates have realized values, so the RCM responses are generated according to (5) using the algorithm given in Section 2.

Table 5 shows the results of our computations on two problem sizes: $(k = 50, n = 500)$ and $(k = 98, n = 1000)$. For each problem size, two cases are shown which correspond to the two likelihood maximizations that need to be computed. In each case results from three runs are shown: one for our TAO code in serial, one for our TAO code in parallel, and one for a simple implementation in R using the `optim` function with the built-in BFGS optimization method. The maximized log-likelihood, walltime in hours:minutes:seconds format, and number of iterations are shown for each run. The number of iterations for serial R was limited to 100, which is the default setting.

For each parallel TAO run, we choose a moderately sized p which evenly divides the number of parameters, and used the smallest number of computing nodes possible to run each job. All parallel TAO jobs shown in the Table 5 were run on either two or three nodes, and 11 to 20 processes overall. As in Section 4, the number of parameters for both problem sizes here were chosen deliberately for convenience, so that the work could be split evenly across a moderate number of processes.

Table 5: Results for LRT computations of generated application problems. For problem size (a), the parallel full space run used 13 processes across 2 nodes, and the parallel restricted run used 17 processes across 2 nodes. For problem size (b), the parallel full space run used 20 processes across 3 nodes, and the parallel restricted run used 11 processes across 2 nodes.

(a) Results for $k = 50, n = 500$

case	#params	run	log-lik	walltime	#iters
Under H_0	51	serial R	-40791.06	02:53:43	57
		serial TAO	-40791.06	00:03:14	14
		parallel TAO	-40791.06	00:00:16	14
Full space	52	serial R	-37284.64	03:21:25	60
		serial TAO	-37284.63	00:05:53	25
		parallel TAO	-37284.63	00:00:35	25

(b) Results for $k = 98, n = 1000$

case	#params	run	log-lik	walltime	#iters
Under H_0	99	serial R	-111241.20	48:54:17	100
		serial TAO	-111241.23	01:12:15	24
		parallel TAO	-111241.23	00:07:43	24
Full space	100	serial R	-104468.55	49:10:42	100
		serial TAO	-104467.38	02:02:48	40
		parallel TAO	-104467.38	00:07:20	40

Notice first that for each case, the log-likelihoods attained across all three runs are nearly the same. This gives some assurance that the TAO and R codes have implemented the problem correctly. The iteration counts match between serial and parallel TAO runs, but R required significantly more iterations. In fact, in the larger problem the iteration limit of 100 has been reached, so more improvement may have been possible. Also, notice that in the restricted case of the larger problem, R has managed to find a slightly better solution than TAO. These issues are not necessarily cause for alarm, since there may be differences between the two optimization methods and their implementations.

Next we see that the R code is dramatically slower than the serial TAO code. For instance, the larger problem required over two days in R to solve either case, whereas the serial TAO code required only about 1 to 2 hours. The R code was not carefully tuned for performance however, so there is likely room for improvement. It would be possible, for example, to create a hybrid R program with the objective function $h(\boldsymbol{\theta})$ written in C. We should also note that our TAO code was not carefully tuned for performance either, so this improvement may not be atypical for what one might see when porting an MLE problem from R to C/C++. Next, moving from serial TAO to parallel TAO, we see that either case of the larger problem can now be solved in about 7 to 8 minutes using at most 20 processes on 3 nodes. Therefore computing the entire LRT for the larger problem has taken about 4 days in serial R, compared to about 3:15 hours in serial TAO, and about 15 minutes in parallel TAO. From Section 4, we would expect performance to scale well with additional parallel processes.

Finally, computing the LRTs using (6) and the TAO results from Table 5 yields

$$-2\log\Lambda = 7012.87, \quad -2\log\Lambda = 13547.70,$$

respectively for the smaller and larger problems. If we believe the approximation $-2\log\Lambda \simeq \chi^2_1$ holds, then we may correctly reject H_0 in both cases and conclude that the number of visits x_i to a counselor has a significant effect on the log-odds of "following the leader".

In this problem we considered a linear model linked to ρ_i, and performed an inference on the model. For demonstration purposes, we constructed a simple model on a single covariate, but let the number of categories in the response be large. Considering a model with many covariates and perhaps a smaller number of categories also leads to a many-parameter situation, where parallel computing would be useful for computing the MLE.

Notice that covariates can easily be added for the counselor, or perhaps at a less granular level such as the high school which employs the counselor, or their geographical region. Covariates at the student level are more granular however, and would complicate the model significantly. Adding covariates at the student level would require the $\boldsymbol{T}_i$'s to be split into smaller observations sharing common counselors, and hence the observations would no longer be independent.

6 Conclusions

We have demonstrated the effectiveness of computing MLEs in parallel using the Random-Clumped model as a test problem. TAO provided an environment to conduct numerical optimizations in parallel, requiring only an objective function, gradient vector, and Hessian matrix. The MLE procedure is just one example of an application that can benefit from this kind of parallel optimization.

We set up an experimental process consisting of random data generation, estimation, and a comparison of the estimate to the true parameters. The effects of adjusting the variables of this experiment were studied. We verified that increasing the number of repetitions or samples increases run time linearly, but increases the quality of the estimates. Increasing the number of multinomial categories causes optimization to become more difficult. Increasing the cluster size increases run time sub-linearly, and improves the quality of our estimates.

We have also studied the parallel performance, varying the number of processes along with sample size, number of multinomial categories, and cluster size. We observed excellent parallel performance when varying sample size and cluster size. The best parallel performance is possible when the number of categories is large. However, increasing the number of categories causes run time to increase faster than linearly. Therefore problems with a very large number of categories will be infeasible to solve even on a large cluster, using the basic method presented here. Smaller experiments with large numbers of repetitions can be solved without the use of a high performance computing cluster. Multiple repetitions are computationally independent, so little communication is needed between processes to compute them in parallel. This kind of parallelism can be accomplished with less elaborate programming, using tools like the SNOW package for R.

Finally, in our application we saw how the parallel MLE method could be used in a more realistic RCM analysis. We used this method to compute the numerator and denominator of several LRTs, to conduct a test for the slope in an embedded linear model. This yielded a significant improvement in performance using only a few computing nodes.

There are many opportunities for future work. The approach used here can be applied to statistical computations in general; particularly we have exploited it to compute MLEs for the Random-Clumped model. For this model, useful theoretical results are available to

vastly improve the performance of MLE computation. For example, Neerchal and Morel [9] suggest a block diagonal approximation for the Fisher Information matrix, which can be used to effectively carry out Fisher Scoring iterations. Further improvements are proposed in [7], in the context of EM. Incorporating these results could yield vastly improved performance, and perhaps new opportunities for applications of the RCM model.

Acknowledgments

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See http://www.umbc.edu/hpcf for more information on HPCF and the projects using its resources. The first author additionally acknowledges financial support as HPCF RA.

References

[1] G.H. Givens and J.A. Hoeting, *Computational Statistics*, Wiley-Interscience, 2005.

[2] A.M. Raim and M.K. Gobbert, *Parallel Performance Studies for a Maximum Likelihood Estimation Problem Using TAO*, HPCF–2009–8, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2009.

[3] S. Benson, L.C. McInnes, J. Moré, T. Munson, and J. Sarich, *TAO User Manual (Revision 1.9)*, ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory, 2007 http://www.mcs.anl.gov/tao.

[4] R. Malouf, *A comparison of algorithms for maximum entropy parameter estimation*, in *COLING-02: Proceedings of the 6th Conference on Natural Language Learning*, Association for Computational Linguistics, 2002, pp. 1–7.

[5] J.G. Morel and N.K. Nagaraj, *A finite mixture distribution for modelling multinomial extra variation*, Biometrika 80 (1993), pp. 363–371.

[6] A. Rossini, L. Tierney, and N. Li, *Simple Parallel Statistical Computing in R*, Working Paper 193, UW Biostatistics Working Paper Series, 2003 http://www.bepress.com/uwbiostat/paper193.

[7] M. Liu, *Estimation for Finite Mixture Multinomial Models*, PhD Thesis, University of Maryland, Baltimore County, Department of Mathematics and Statistics, 2005.

[8] T. Banerjee and S. Paul, *Miscellanea. An extension of Morel-Nagaraj's finite mixture distribution for modelling multinomial clustered data*, Biometrika 86 (1999), pp. 723–727.

[9] N.K. Neerchal and J.G. Morel, *Large Cluster Results for Two Parametric Multinomial Extra Variation Models*, Journal of the American Statistical Association 93 (1998), pp. 1078–1087.

[10] H. Zhou and K. Lange, *MM Algorithms for Some Discrete Multivariate Distributions*, Journal of Computational and Graphical Statistics 19 (2010), pp. 645–665.

[11] N.K. Neerchal and J.G. Morel, *An improved method for the computation of maximum likelihood estimates for multinomial overdispersion models*, Computational Statistics & Data Analysis 49 (2005), pp. 33–43.

[12] M.K. Gobbert, *Parallel Performance Studies for an Elliptic Test Problem*, HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.