Parallel Performance Studies for a Maximum Likelihood Estimation Problem Using TAO

Andrew M. Raim and Matthias K. Gobbert
Department of Mathematics and Statistics, University of Maryland, Baltimore County

Abstract

In this report, we present an application of parallel computing to an estimation procedure in statistics. The method of maximum likelihood estimation (MLE) is based on the ability to perform maximizations of probability functions. In practice, this work is often performed by computer with numerical methods, and may be time consuming for some likelihood functions. We consider one such likelihood function based on the Finite Mixture Multinomial distribution. We perform estimation for this problem in parallel using the Toolkit for Advanced Optimization (TAO) software library. The computations are performed on a distributed-memory cluster with InfiniBand interconnect in the High Performance Computing Facility at University of Maryland, Baltimore County (UMBC). We study how the resource requirements change as problem sizes vary, and demonstrate that scaling the number of processes for larger problems decreases wall clock time significantly.

1 Introduction

Consider a random sample of n observations $\boldsymbol{X} = (X_1, \ldots, X_n)$ from a probability distribution $f(x \mid \boldsymbol{\theta})$. That is, $X_1, \ldots, X_n$ are independently and identically distributed random variables and each distributed according to $f(x \mid \boldsymbol{\theta})$. The form of this probability function is assumed to be known up to a vector of parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_k)$. The unknown parameter $\boldsymbol{\theta}$ belongs to the space Ω, a subset of the Euclidean space of appropriate dimension. The process of obtaining a plausible value of the parameter that best suits the observed data is known as the statistical point estimation. One common method of estimation is the *method of maximum likelihood*, which is based on the paradigm that the value of $\boldsymbol{\theta}$ which maximizes the likelihood function is proposed as an estimate. For a given dataset $\boldsymbol{x} = (x_1, \ldots, x_n)$, the likelihood function, namely the joint probability function evaluated at the observed data, is given by

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

and thus the maximum likelihood estimate (MLE) is defined as

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

Typically, we maximize the logarithm of the likelihood function, known as the log-likelihood function,

$$\ell(\boldsymbol{\theta} \mid \boldsymbol{x}) \stackrel{\mathrm{def}}{=} \log L(\boldsymbol{\theta} \mid \boldsymbol{x}) = \sum_{i=1}^{n} \log L(\boldsymbol{\theta} \mid x_i).$$

For simple situations, the maximization can be performed analytically. However, more complicated scenarios require use of numerical methods. Some commonly used numerical methods include expectation maximization, Newton-Raphson, and Fisher Scoring. See [GH05] for a good overview.

We investigate the computation of maximum likelihood estimates using a parallel architecture. In particular, we use a software library TAO (Toolkit for Advanced Optimization) [BMM+07], which implements a number of commonly used numerical optimization methods. TAO is a special optimization library, because it is designed for use in parallel computing environments. The objective of this report is to study the effectiveness of applying parallel optimization to the maximum likelihood estimation procedure. We focus specifically on the *limited-memory, variable-metric* (LMVM) unconstrained optimization method. Malouf [Mal02] demonstrates 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 this report, LMVM is used to compute maximum likelihood estimates for the Finite Mixture Multinomial (FMM) distribution described in [MN93], for which numerical optimization is necessary.

In Section 2 we introduce the Finite Mixture Multinomial (FMM) model, and describe how to generate observations from it. In Section 3 we discuss our application of TAO to computation of the MLE. In Section 4 we design an experiment consisting of sample generation and estimation to measure the performance of the implementation. 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 and that the parallel performance is very good overal. Finally concluding remarks are given in Section 5.

2 The Test Problem

First, let us consider the standard multinomial distribution, which arises in a natural way when a group of m people are asked a question with k possible responses. The set of possible outcomes $\mathbb{S}$ consists of vectors of counts, representing how many participants have chosen each response. $\mathbb{S}$ is a discrete set, which can be shown to contain $\binom{n+k-1}{k}$ elements. Let $\boldsymbol{T} = (T_1, \ldots, T_k)$ denote the vector of counts, where T_i denotes the number of people who gave the ith response. Clearly, $\sum_{i=1}^{k} T_i = m$. If we assume that the participants respond to the question independently of each other, the vector will be distributed according to the multinomial distribution. The probability distribution function for the multinomial distribution is

$$f(\boldsymbol{t} \mid \eta_1, \ldots, \eta_k, m) = \frac{m!}{t_1!\, t_2! \cdots t_k!}\, \eta_1^{t_1}\, \eta_2^{t_2} \cdots \eta_k^{t_k}. \tag{2.1}$$

The parameters $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_k)$ are the probabilities that a participant in the survey will choose the corresponding responses. This means that $\sum_{i=1}^{k} \eta_i = 1$ and $0 \le \eta_i \le 1$, and therefore only $k-1$ parameters will need to be estimated — the kth is redundant information. We should also notice that m can be observed directly from the data, and that $T_k = m - \sum_{i=1}^{k-1} T_i$. If a random vector $\boldsymbol{T}$ follows this distribution, we write $\boldsymbol{T} \sim \mathrm{M}_k(\eta_1, \ldots, \eta_k, m)$. We 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 vector of counts $\boldsymbol{X} = (\boldsymbol{T}_1, \ldots, \boldsymbol{T}_n)$, which can be visualized as a matrix

$$\boldsymbol{X} = \begin{bmatrix} \begin{pmatrix} T_{1,1} \\ T_{2,1} \\ \vdots \\ T_{k,1} \end{pmatrix} & \begin{pmatrix} T_{1,2} \\ T_{2,2} \\ \vdots \\ T_{k,2} \end{pmatrix} & \cdots & \begin{pmatrix} T_{1,n} \\ T_{2,n} \\ \vdots \\ T_{k,n} \end{pmatrix} \end{bmatrix}.$$

Mixture of Multinomials Mixture distributions involve linear combinations of individual probability functions (see for example [HMC04] for an overview). Thus, a mixture of q multinomials constructed with $\boldsymbol{w} = (w_1, \ldots, w_q)$ is given by

$$f(x \mid \boldsymbol{\theta}) = \sum_{i=1}^{q} w_i\, f_i(x \mid \eta_{i1}, \ldots, \eta_{ik}), \tag{2.2}$$

where $\sum_{i=1}^{q} w_i = 1$, $w_i > 0$ for $i = 1, \ldots, q$, and $\boldsymbol{\eta}_i = (\eta_{i1}, \ldots, \eta_{ik})$ is the vector of probabilities corresponding to the ith component of the mixture. The motivation for considering a mixture distribution may be drawn from the point of view of classification. Suppose the people answering a multinomial response survey are drawn from one of q 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 q independently distributed multinomial count vectors. Since the labels are not available, the likelihood will be the mixture distribution given above. This distribution has been widely used in a number of applications including text mining, linguistics, and clustering. See [Liu05] for a detailed review.

As test problem for our exploration, we consider a special multinomial model using the mixture concept proposed by Morel and Nagaraj [MN93], referred to as the Finite Mixture Multinomial (FMM) model. The model is also described in detail in [NM98]. The motivation for FMM 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 not tenable. Therefore, 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. Morel and Nagaraj [MN93] provide a model for such phenomena (known in the statistics literature as *overdispersion*), which turns out to be a special case of the multinomial mixture distribution given above. In subsequent work [NM98, NM05], they show that the model has many desirable theoretical and practical properties.

The FMM distribution can be motivated by correlating responses obtained 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

$$\begin{aligned} N &\sim \text{Binomial}(\rho, m), \\ \boldsymbol{Y} &\sim M_k(\Pi, 1), \\ \text{and } (\boldsymbol{X} \mid N) &\sim M_k(\Pi, m - N). \end{aligned}$$

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

$$f(\boldsymbol{t} \mid \Pi, \rho) = \sum_{i=1}^{k} \pi_i \, g(\boldsymbol{t} \,|\, \boldsymbol{\eta}_i, m),$$

where

$$\begin{aligned} &g(\boldsymbol{t} \,|\, \boldsymbol{\eta}_i, m) \text{ is the probability distribution function of a standard multinomial,} \\ &\boldsymbol{\eta}_i = (1 - \rho)\Pi + \rho \, \boldsymbol{e}_i, \quad i = 1, 2, \ldots, k - 1, \\ &\boldsymbol{\eta}_k = (1 - \rho)\Pi, \end{aligned}$$

where $\boldsymbol{e}_i$ is a ith unit vector with 1 in the ith position and 0 in all other positions. Hence, we obtain FMM as a special case of the mixture distribution of (2.2). Therefore, there are k distinct parameters $\boldsymbol{\theta} = (\pi_1, \ldots, \pi_{k-1}, \rho)$ in the FMM distribution. The joint likelihood function for n observations $\boldsymbol{X} = (\boldsymbol{t_1}, \ldots, \boldsymbol{t_n})$ is

$$L(\boldsymbol{\theta} \mid \boldsymbol{X}) = \prod_{i=1}^{n} L(\boldsymbol{\theta} \mid \boldsymbol{t}_i) = \prod_{i=1}^{n} f(\boldsymbol{t} \mid \Pi, \rho). \tag{2.3}$$

Our objective is to compute $\widehat{\boldsymbol{\theta}}_{\text{MLE}}$ for these parameters, using randomly generated data from the distribution. Although we will not make use of them in this paper, theoretical results are available in [NM05] and [Liu05] that help to make the computations more manageable.

Generating Sample Data Neerchal and Morel [NM05] describe a method for generating random samples from the FM multinomial 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 $(\eta_1, \ldots, \eta_k)$. Generate m observations from the uniform distribution $u_1, \ldots, u_m \overset{iid}{\sim} \mathrm{U}(0,1)$. For observation u_j, determine the category c such that

$$\eta_1 + \cdots + \eta_{c-1} < u_j \leq \eta_1 + \cdots + \eta_c$$

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

To generate samples from the FMM distribution, begin with known parameters $(\pi_1, \ldots, \pi_k, \rho)$. Generate $m+1$ observations from the standard multinomial distribution $\boldsymbol{s}, \boldsymbol{s}_1^0, \ldots, \boldsymbol{s}_m^0 \overset{iid}{\sim} \mathrm{M}(\pi_1, \ldots, \pi_k, 1)$, and m observations from the uniform distribution $u_1, \ldots, u_m \overset{iid}{\sim} \mathrm{U}(0,1)$. The entries of the new FM multinomial observation are given by

$$\boldsymbol{t}_i = \boldsymbol{s}\, I(u_i \leq \rho) + \boldsymbol{s}_i^0\, I(u_i > \rho),$$

where I represents the indicator function. Availability of this simple and intuitive algorithm of generating data is one of the many reasons for the choice of FMM as our test problem. Others are, it is identifiable for all values of $(\pi_1, \ldots, \pi_k, \rho)$ and does not require additional assumptions for identifiability. Furthermore, it essentially encompasses all numerical issues one may face in the MLE of the full mixture.

3 Numerical Method Implemented

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 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 are 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 the numerical method. Using the libraries this way however results in a loss of control, which could be detrimental to code performance. TAO provides a suite of optimization algorithms and a framework to use them. The programmer provides several important ingredients such as the objective function $h(\boldsymbol{x})$ to optimize, a function to compute its gradient vector $\nabla h(\boldsymbol{x})$, and a function to compute its Hessian matrix $H(\boldsymbol{x})$. Our implementation is carried out in the C++ programming language. The centerpiece is the objective function $h(\boldsymbol{\theta})$, the log-likelihood function from (2.3)

$$h(\boldsymbol{\theta}) \overset{\text{def}}{=} -\ln\left[\prod_{i=1}^{n} L(\boldsymbol{\theta} \mid \boldsymbol{t}_i)\right] = -\sum_{i=1}^{n} \ln L(\boldsymbol{\theta} \mid \boldsymbol{t}_i). \tag{3.1}$$

TAO algorithms perform minimizations, so a negative sign is applied to achieve maximization. We also choose to work with the log-likelihood, which involves a summation rather than a product, because products of many probabilities could involve floating point numbers of very small magnitude.

Computation of $h(\boldsymbol{\theta})$ involves the evaluation of the standard multinomial distribution (2.1), which involves the factorial function. Computing factorials naively can lead to serious numerical problems. Although the final result is a probability between 0 and 1, results of intermediate factorial calculations may be too large to store as double precision floating point numbers. Again we use logarithms to stabilize the calculation

$$\ln\left[\frac{m!}{t_1!\,t_2!\,\cdots\,t_k!}\,\eta_1^{t_1}\,\eta_2^{t_2}\,\cdots\,\eta_k^{t_k}\right] = \ln(m!) - \sum_{i=1}^{k}\ln(t_i!) + \sum_{i=1}^{k} t_i \ln(\eta_i). \tag{3.2}$$

For individual terms of the form $\ln(t!)$, we use the function $\ln[\Gamma(t+1)]$ instead of calculating factorials explicitly. Recall that the Γ function is defined such that $\Gamma(t) = (t-1)\Gamma(t-1)$ for positive integers, and therefore $t! = \Gamma(t+1)$. The C math library contains functions such as `lgamma` and `lgamma_r` which perform this computation, and are assumed to be optimal. Taking the exponential of (3.2) yields (2.1), as desired.

Optimization Methods Several algorithms are available in TAO for unconstrained optimization (minimization), which are described in [BMM+07]. The algorithms operate by evaluating the objective function, the gradient vector, and the Hessian matrix during the search for the optimal solution. It is the responsibility of the programmer to provide functions to compute the gradient vector and Hessian matrix, given a candidate solution.

The Nelder-Mead (NM) method is typically the worst performer, but requires only an objective function. The nonlinear conjugate gradient (CG) method and limited-memory, variable-metric (LMVM) method require both an objective function and a gradient function to be implemented. The Newton Line Search (NLS) method uses the Hessian matrix in addition, and is also typically the best performer of the methods mentioned. For the experiments conducted in this report, we consider only LMVM[1].

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

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

where $H^{(i)}$ is an approximation to the Hessian computed within the method. This computation utilizes a limited amount of information from the previous steps, hence the name "limited-memory". After the direction is obtained, a line search is performed to compute the size of the next step τ, such that

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

is minimized.

The TAO/PETSc framework is quite flexible and allows many configuration options to be specified on the command line. Solvers and preconditioners can be selected, and tuning parameters of the algorithms can be specified. We use several options:

- `-tao_method tao_lmvm` — select the LMVM method
- `-options_table` — print the list of options after executing
- `-tao_ls_maxfev 99999999` — set the maximum number of function evaluations in a line search to a high number
- `-tao_max_its 99999999` — set the maximum number of iterations to a high number
- `-tao_max_funcs 99999999` — set the maximum number of function evaluations to a high number
- `-tao_view` — print diagnostic information about the solver after optimization is complete

We leave all the tuning parameters at their defaults. There are also options which are helpful for debugging. These include enabling step-by-step logging in TAO and PETSc, and viewing the solution, gradient, and Hessian after each iteration.

[1] In preliminary tests, the other methods suffered from inferior performance and convergence problems when problem sizes started to scale. This was likely an issue with our implementation, and not with the library or the methods themselves.

Enforcing Constraints Unconstrained optimization algorithms cannot be directly applied to our problem, because our parameters $\boldsymbol{\theta} = (\pi_1, \ldots, \pi_k, \rho)$ are all probabilities which must lie between 0 and 1. To enforce this constraint we make use of the logit transformation, as suggested in [Lin01],

$$\text{logit}(x) = \frac{1}{1 + e^{-x}},$$

which maps $x \in \mathbb{R}$ to the interval $(0, 1)$.

The other important constraint is that $\sum_{i=1}^{n} \pi_i$ must be equal to 1. To enforce this, we normalize the vector of π_i's by scaling all components with $\sum_{i=1}^{n} \pi_i$. Therefore to evaluate $h(\boldsymbol{\theta})$ with a candidate solution $\widehat{\boldsymbol{\theta}}^{(i)}$ from the optimizer, we first apply the logit function to each entry of $\widehat{\boldsymbol{\theta}}^{(i)}$, then apply the above normalization. Because of the required normalization step, we cannot use the fact that $\pi_k = 1 - \sum_{i=1}^{k-1} \pi_i$ to infer π_k. We must find each of the π_i parameters explicitly in the optimization, and therefore have $k + 1$ total parameters under consideration.

Gradient Evaluation We use finite differences to numerically evaluate the gradient vector

$$\nabla h(\boldsymbol{\theta}) = \frac{\partial h(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \left(\frac{\partial h(\boldsymbol{\theta})}{\partial \theta_1}, \ldots, \frac{\partial h(\boldsymbol{\theta})}{\partial \theta_{k+1}} \right).$$

Each entry in the vector can be computed using the approximation

$$\frac{\partial h(\boldsymbol{\theta})}{\partial \theta_i} = \lim_{d \to 0} \left(\frac{h(\boldsymbol{\theta} + d\boldsymbol{e}_i) - h(\boldsymbol{\theta})}{d} \right) \approx \frac{h(\boldsymbol{\theta} + \delta \boldsymbol{e}_i) - h(\boldsymbol{\theta})}{\delta}$$

where $\boldsymbol{e}_i$ is the ith unit vector with 1 in the ith position and 0 in all other positions. We choose δ to be 10^{-8} for our approximations. Clearly, $k + 1$ elements of $\nabla h(\boldsymbol{\theta})$ must be computed each time the gradient is evaluated.

Hessian Evaluation We describe the finite difference method for computing the Hessian matrix for reference, even though it is not used in LMVM. The definition of the Hessian is

$$H(\boldsymbol{\theta}) = \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^2} = \begin{bmatrix} \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_1} & \cdots & \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_{k+1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \theta_{k+1} \partial \theta_1} & \cdots & \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \theta_{k+1} \partial \theta_{k+1}} \end{bmatrix}.$$

Each entry in the matrix can be computed using the approximation

$$\begin{aligned} \frac{\partial^2 h(\boldsymbol{\theta})}{\partial \theta_i \theta_j} &= \lim_{d_1 \to 0, d_2 \to 0} \left(\frac{h(\boldsymbol{\theta} + d_1 \boldsymbol{e}_i + d_2 \boldsymbol{e}_j) - h(\boldsymbol{\theta} + d_1 \boldsymbol{e}_i) - h(\boldsymbol{\theta} + d_2 \boldsymbol{e}_j) + h(\boldsymbol{\theta})}{d_1 \, d_2} \right) \\ &\approx \frac{h(\boldsymbol{\theta} + \delta_1 \boldsymbol{e}_i + \delta_2 \boldsymbol{e}_j) - h(\boldsymbol{\theta} + \delta_1 \boldsymbol{e}_i) - h(\boldsymbol{\theta} + \delta_2 \boldsymbol{e}_j) + h(\boldsymbol{\theta})}{\delta_1 \, \delta_2} \end{aligned}$$

where $\boldsymbol{e}_i$ is the ith unit vector with 1 in the ith position and 0 in all other positions. We choose δ_1 and δ_2 to be 10^{-8} for our approximations. Clearly, $(k+1) \times (k+1)$ elements of $H(\boldsymbol{\theta})$ must be computed each time the matrix is evaluated.

Code Implementation Our use of TAO is based on two principal functions `FormFunctionGradient` and `FormHessian`. Given a candidate solution vector $\boldsymbol{\theta}^{(i)}$, the former function computes the current objective $h(\boldsymbol{\theta}^{(i)})$ and gradient $\nabla h(\boldsymbol{\theta}^{(i)})$ values, and the latter computes the Hessian $H(\boldsymbol{\theta}^{(i)})$. Both functions depend on the FM multinomial likelihood function given in (3.1), which we have implemented in C++. Recall that only

`FormFunctionGradient` is relevant to the experiments in this report, but we discuss `FormHessian` because a more complete use of TAO would involve implementing both.

TAO/PETSc supports several varieties of sparse data structures. For our application, we do not expect sparseness in the gradient or Hessian, so we opt to use the dense vector and dense matrix implementations. In applications involving dense matrices, parallelization in TAO/PETSc is achieved by splitting the matrices by rows across the available p processes. The first process stores the first block of rows locally, the second process stores the second block, etc. This scheme is potentially limiting to the kinds of parallel solutions that can be applied, but also greatly simplifies use of the libraries.

For all experiments in this report, we choose the total number of parameters $k+1$ such that p divides $k+1$ evenly. This is done for convenience and to demonstrate the ideal case of parallel performance for equal load balancing and not a limitation of the method itself. Parallelism in our implementation is achieved by partitioning the vector of parameters. The key observation is that the entries in the gradient vector and Hessian matrix can be computed independently. Computing the likelihood function can be quite expensive, because it requires iterating over each observation in the sample (refer to (3.1)). Therefore, splitting function evaluations evenly among the $p \leq k$ processes should result in good scalability. The candidate solution $\boldsymbol{\theta}^{(i)}$ is split across the p processes, but each process needs the entire current solution to perform any of the necessary computations. Therefore we perform an `MPI_Allgather` operation at the start of `FormFunctionGradient`. Then each process can update its local portion of the gradient and Hessian. Some of the internals of the TAO optimization algorithms potentially work in parallel as well (e.g., linear solves) which would further improve performance, but it is not 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.

4 Computational Experiments and Results

Experiment Structure 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 (the total number of parameters is $k+1$),
- number of repetitions r,
- number of MPI processes p.

To determine the true values for FMM 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 $\mathit{\Pi} = \boldsymbol{v}/\sum_i v_i$, and let $\rho = 1/4$ so that the true parameters of the experiment are $\boldsymbol{\theta} = (\mathit{\Pi}, \rho)$. 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 FMM distribution with these parameters, as described in Section 2. The objective function $h(\boldsymbol{\theta})$ given in (3.1) 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 data matrices $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_r$, and the estimates $\widehat{\boldsymbol{\theta}}^{(1)}_{\text{MLE}}(\boldsymbol{X}_1), \ldots, \widehat{\boldsymbol{\theta}}^{(r)}_{\text{MLE}}(\boldsymbol{X}_r)$.

We can consider the samples $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_r$ as independent and identically distributed random matrices, drawn from the FMM distribution. By a theorem of Morel and Nagaraj [MN93], we have that the MLE is consistent and asymptotically normal for FMM. Therefore we expect that as r increases, the average $\bar{\boldsymbol{\theta}}_{\text{MLE}} = \frac{1}{r}\sum_{i=1}^{r} \widehat{\boldsymbol{\theta}}^{(i)}_{\text{MLE}}(\boldsymbol{X}_i)$ will approach the true value $\boldsymbol{\theta}$. Thus, we create a measurement of solution quality based on the average of the estimates and its distance from the true parameters. 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. We define the sum-squared error between $\bar{\boldsymbol{\theta}}_{\text{MLE}}$ and $\boldsymbol{\theta}$ as

$$\text{SSE}(\bar{\boldsymbol{\theta}}_{\text{MLE}}, \boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_{i=1}^{k+1} \left(\bar{\theta}_{\text{MLE},i} - \theta_i\right)^2 . \tag{4.1}$$

While criterion (4.1) does not guarantee that we are computing the global maxima of the likelihood function, it would provide evidence that the algorithm has converged to a consistent solution of the MLE problem.

Technically, the sample generation process is conducted outside of the experiments. A sample is generated for each distinct setting of (n, k, m), for the maximum number of repetitions which will be needed at that setting. Each sample is stored in a file, using a systematic naming scheme such as `sample_n0016_k127_m064_rep0002.mat`. During the "sample generation" phase in an experiment, the appropriate sample file is found 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 reinitialization of the TAO framework to its initial state between iterations. Walltime does not include calculation of relative 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 is measured by taking the difference between starting and ending measurements. We record the total memory size 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 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 [Gob08] 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. Figures 1, 2, 3, 4, and 5 display the effect on run time and solution quality for each varying parameter. The optimization method used in all runs is LMVM. The column `mem.kb` shows the global memory (in kB) used across all processes, as described earlier. The column `sum.sq.error` is the SSE quantity defined in (4.1). `Iterations` represents the total number of times (over the entire experiment) that `FormFunctionGradient` was invoked by the TAO framework; this number will be the same 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 calculation discussed in Section 3, which is implemented using the `lgamma_r` function in C. This is an indication that the time to compute `lgamma_r` depends on the size of its argument. We can also see that the number of iterations is affected somewhat by the value of m, indicating that the optimization problem becomes more difficult for larger m. 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 increases along with k, but which unexpectedly has not increased between $k = 31$ and $k = 63$. Despite the drop in iteration count, the run

Experiments varying n								
m	k	n	r	p	walltime	mem.kb	sum.sq.error	iterations
32	7	32	16	1	3.976767	264	7.656012e-05	362
32	7	64	16	1	7.758089	264	3.509462e-05	343
32	7	128	16	1	15.900614	264	1.266233e-05	354
32	7	256	16	1	33.357186	300	1.090919e-05	369
32	7	512	16	1	67.875619	456	8.929867e-06	376
32	7	1024	16	1	133.642802	636	5.142452e-06	371
32	7	2048	16	1	269.742766	1216	9.101082e-07	379
32	7	4096	16	1	638.068784	1820	9.615955e-07	446
Experiments varying m								
m	k	n	r	p	walltime	mem.kb	sum.sq.error	iterations
1	31	128	16	1	313.0260	276	1.393184e-01	175
2	31	128	16	1	247.7760	276	6.273286e-02	138
4	31	128	16	1	228.6934	276	6.264830e-02	124
8	31	128	16	1	240.4331	276	6.257713e-02	130
16	31	128	16	1	273.8790	276	6.254580e-02	143
32	31	128	16	1	354.6643	276	2.363615e-02	177
64	31	128	16	1	446.8721	276	1.313185e-05	201
128	31	128	16	1	493.5351	276	4.987544e-06	204
256	31	128	16	1	657.7319	276	3.043573e-06	262
512	31	128	16	1	647.8922	276	1.183818e-06	253
Experiments varying k								
m	k	n	r	p	walltime	mem.kb	sum.sq.error	iterations
256	3	64	16	1	0.436564	264	1.139250e-06	124
256	7	64	16	1	3.765587	264	4.469050e-06	165
256	15	64	16	1	36.885492	276	4.806409e-06	215
256	31	64	16	1	313.977938	284	3.416361e-06	250
256	63	64	16	1	2261.357719	264	3.245782e-06	244
Experiments varying r								
m	k	n	r	p	walltime	mem.kb	sum.sq.error	iterations
32	7	128	1	1	1.012896	264	5.183138e-04	23
32	7	128	2	1	1.971573	264	4.082971e-04	46
32	7	128	4	1	3.977437	264	1.262012e-04	90
32	7	128	8	1	7.989043	264	7.135169e-05	179
32	7	128	16	1	15.633417	264	1.266233e-05	354
32	7	128	32	1	31.525529	264	1.168983e-05	711
32	7	128	64	1	63.706299	264	1.348642e-05	1428
32	7	128	128	1	129.828254	268	6.337458e-06	2881
Experiments varying p								
m	k	n	r	p	walltime	mem.kb	sum.sq.error	iterations
256	31	256	16	1	1338.8958	516	8.648826e-07	267
256	31	256	16	2	681.8996	524	8.648537e-07	267
256	31	256	16	4	351.3953	468	8.648611e-07	267
256	31	256	16	8	188.3969	1036	8.649601e-07	267
256	31	256	16	16	106.9965	1244	8.648233e-07	267
256	31	256	16	32	68.0193	4296	8.647875e-07	267

Table 1: Results for consistency experiments.

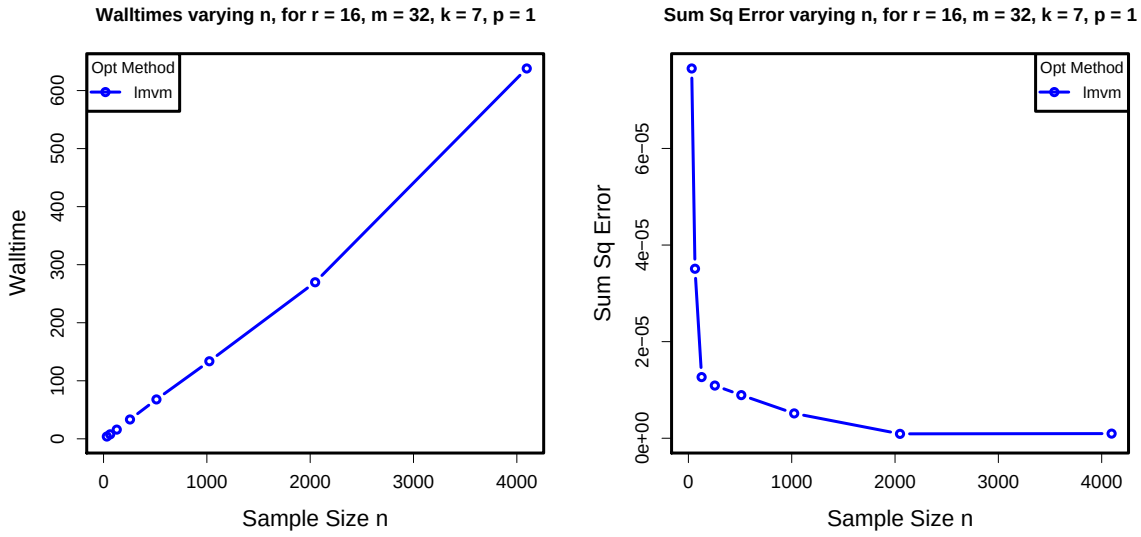

Figure 1: Consistency experiments — plots showing the effect of n on walltime and solution quality.

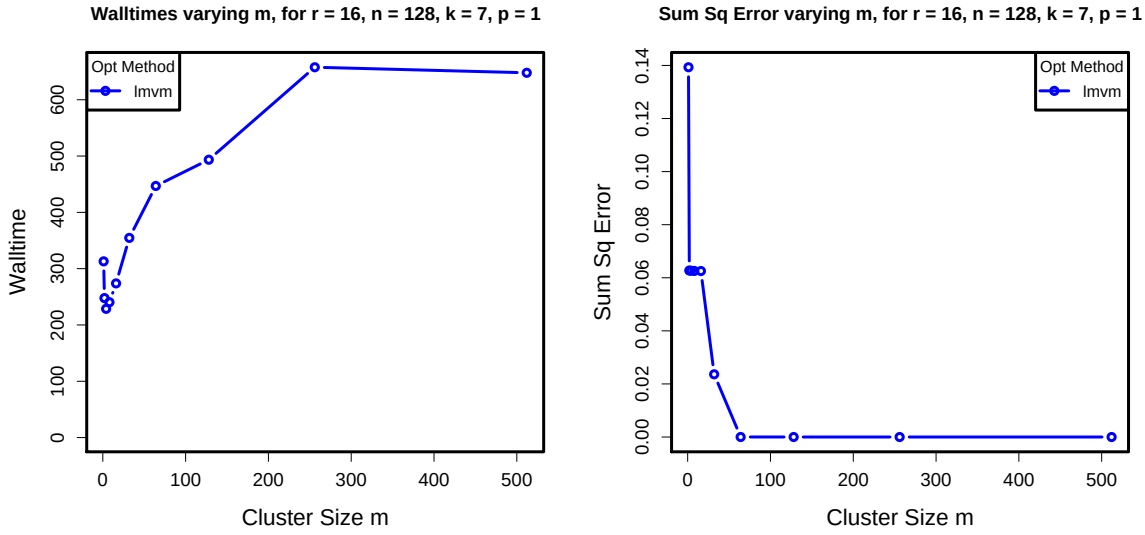

Figure 2: Consistency experiments — plots showing the effect of m on walltime and solution quality.

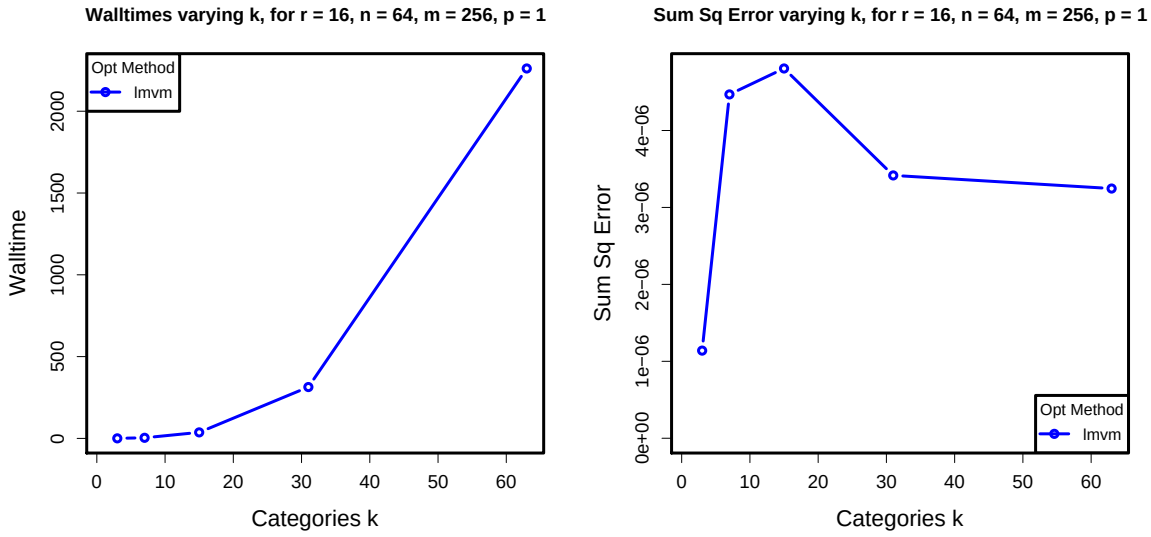

Figure 3: Consistency experiments — plots showing the effect of k on walltime and solution quality.

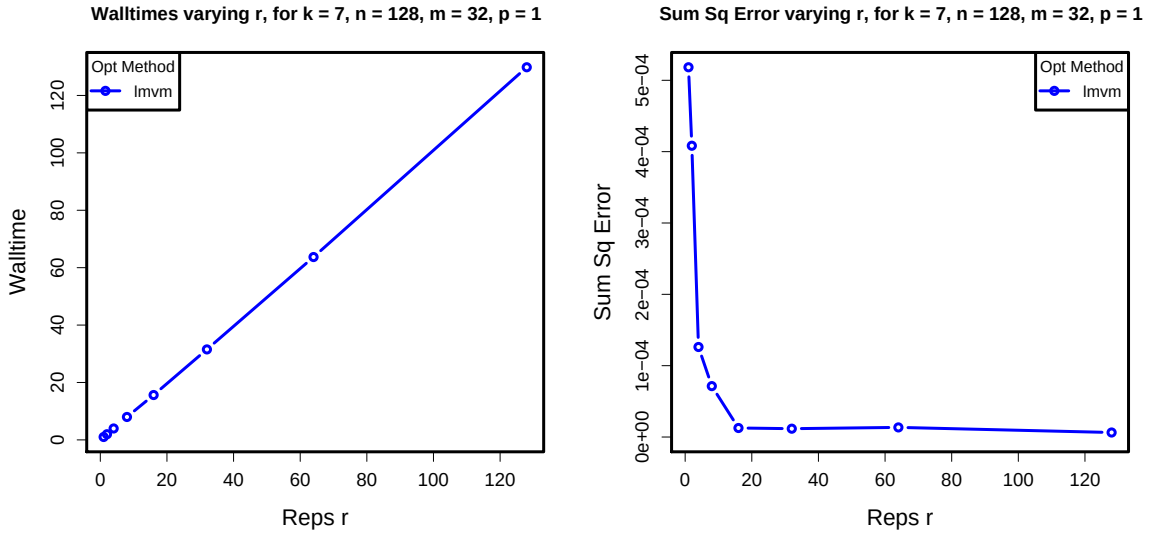

Figure 4: Consistency experiments — plots showing the effect of r on walltime and solution quality.

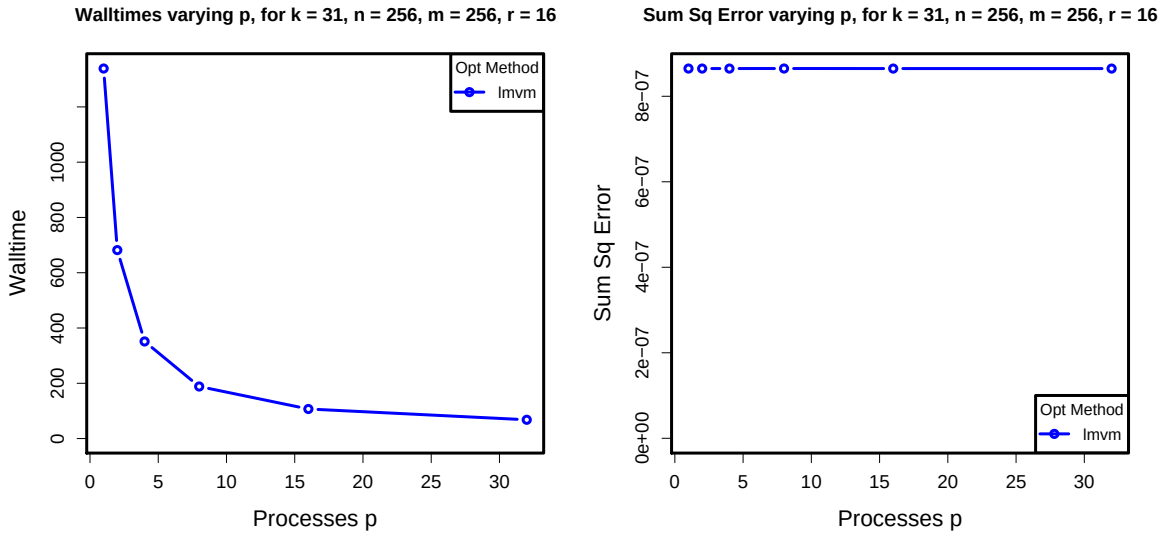

Figure 5: Consistency experiments — plots showing the effect of p on walltime and solution quality.

time has still increased substantially between $k = 31$ and $k = 63$ because the time to evaluate the objective function depends on k. 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.

4.2 Performance Experiments

We consider the parallel performance of the FMM 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 the performance of multiple repetitions. We examine walltime as well as the metrics speedup and efficiency, which are defined as follows. 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 6 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 7 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 8 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 their corresponding single process runs, to ensure correct computations in parallel (these results are not shown in this report). We found that nearly all settings of p result in the same solution, up to rounding error. For several cases, one setting of p resulted in a solution vector which was slightly different and an iteration count one less than the other settings of p. The final objective function value was correspondingly slightly larger in this case. We note that this can easily happen in practical calculations, and it is useful to observe that the performance remains excellent despite the slightly different iteration count.

5 Conclusions

We have demonstrated the use of TAO to perform maximum likelihood estimation for the FM multinomial distribution problem. The MLE procedure is just one example of an application that can benefit from 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 our 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 verified that the results of multiple process runs are close to the results of single process runs, and hence we have some confidence in the quality of the results. 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 standard method presented here. Smaller experiments with large numbers of repetitions can be solved without the use of a high performance computing cluster. The repetitions are computationally independent, so little communication is needed between processes. This kind of parallelism can be accomplished with less intricate programming work, using tools like the Simple Network of Workstations (snow, http://www.sfu.ca/~sblay/R/snow.html) package for R.

The are many opportunities for future work. We have only studied the performance of the LMVM optimization method in TAO, but the NLS method is potentially a better choice for unconstrained optimization. There are also numerous constrained methods that may be applicable. Several theoretical results are available for the FM multinomial distribution to improve overall performance of computations. Neerchal and Morel [NM98] suggest a block diagonal approximation for the expectation of the Hessian matrix. Further improvements are proposed in [Liu05]. Incorporating these results could yield vastly improved performance, and perhaps new opportunities for parallelization.

Walltime (sec)								
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
Speedup								
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
Efficiency								
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

Table 2: Walltime, speedup, and efficiency varying n, for $k = 127$, $m = 64$, $r = 1$.

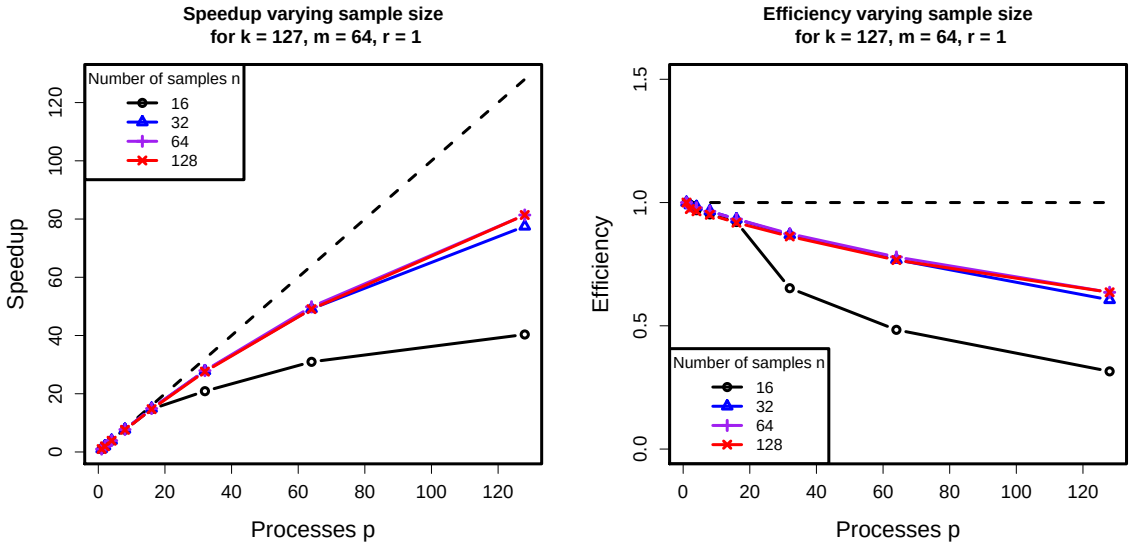

Figure 6: Scalability experiments — Plots showing scalability as n varies. The $n = 64$ series is hidden underneath $n = 128$.

Walltime (sec)								
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
Speedup								
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
Efficiency								
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

Table 3: Walltime, speedup, and efficiency varying m, for $n = 128$, $k = 127$, $r = 1$.

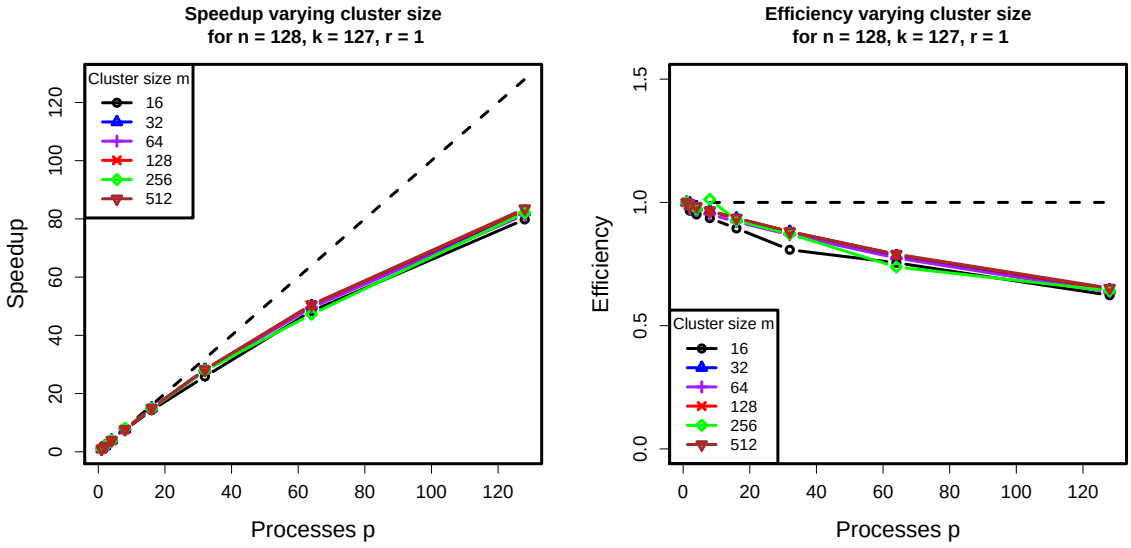

Figure 7: Scalability experiments — Plots showing scalability as m varies.

Walltime (sec)								
k	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
1	0.0042	0.0048	—	—	—	—	—	—
3	0.0576	0.0361	0.0252	—	—	—	—	—
7	0.4708	0.2549	0.1483	0.1514	—	—	—	—
15	4.9128	2.5070	1.3746	0.8244	0.5993	—	—	—
31	41.7236	21.4142	11.0095	5.9119	3.3939	5.1653	—	—
63	390.4026	197.0001	99.9619	51.8082	27.5437	14.7214	9.0632	—
127	2513.4461	1259.7158	635.5848	306.8063	167.3951	92.0081	49.7095	30.3673
Speedup								
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
Efficiency								
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

Table 4: Walltime, speedup, and efficiency varying k, for $n = 128$, $m = 256$, $r = 1$.

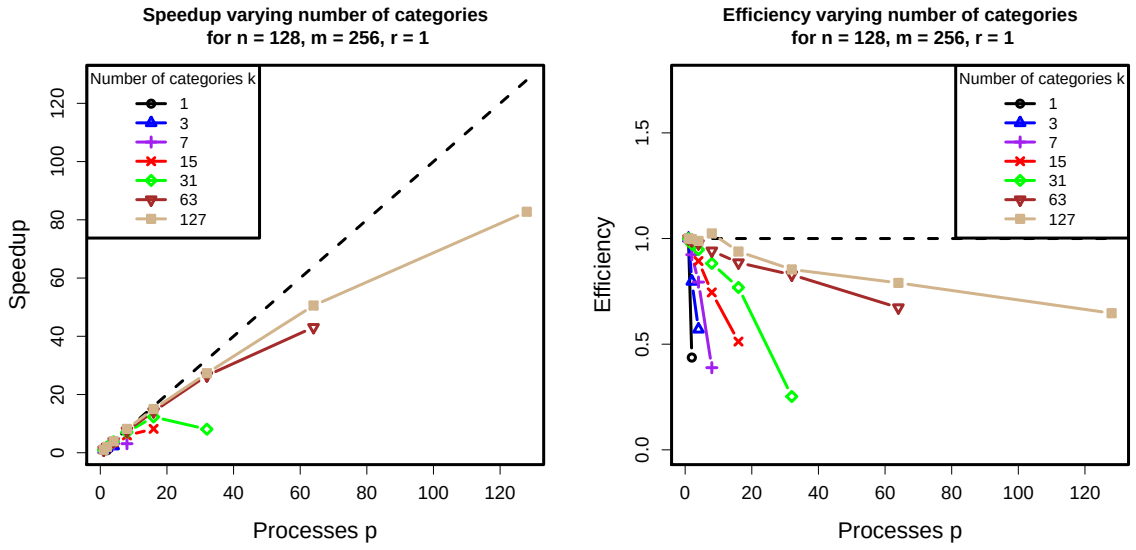

Figure 8: Scalability experiments — Plots showing scalability as k varies.

Acknowledgments

The authors would like to thank Dr. Nagaraj K. Neerchal for his numerous contributions to this work. These include offering the Finite Mixture Multinomial model as a test problem, many fruitful discussions, and substantial input during the reviewing and editing processes. 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.

References

[BMM+07] Steve Benson, Lois Curfman McInnes, Jorge Moré, Todd Munson, and Jason Sarich. TAO user manual (revision 1.9). Technical Report ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory, 2007. http://www.mcs.anl.gov/tao.

[GH05] Geof H. Givens and Jennifer A. Hoeting. *Computational Statistics*. Wiley-Interscience, 2005.

[Gob08] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

[HMC04] Robert V. Hogg, Joseph W. McKean, and Allen Craig. *Introduction to Mathematical Statistics*. Prentice Hall, 6th edition, 2004.

[Lin01] James K. Lindsey. *Nonlinear Models in Medical Statistics*. Oxford University Press, 2nd edition, 2001.

[Liu05] Minglei Liu. *Estimation for Finite Mixture Multinomial Models*. PhD thesis, University of Maryland, Baltimore County, 2005.

[Mal02] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *COLING-02: Proceedings of the 6th Conference on Natural Language Learning*, pages 1–7. Association for Computational Linguistics, 2002.

[MN93] Jorge G. Morel and Neerchal K. Nagaraj. A finite mixture distribution for modelling multinomial extra variation. *Biometrika*, 80(2):363–371, 1993.

[NM98] Nagaraj K. Neerchal and Jorge G. Morel. Large cluster results for two parametric multinomial extra variation models. *Journal of the American Statistical Association*, 93(443):1078–1087, 1998.

[NM05] Nagaraj K. Neerchal and Jorge G. Morel. An improved method for the computation of maximum likelihood estimates for multinomial overdispersion models. *Computational Statistics & Data Analysis*, 49(1):33–43, 2005.