Generalized Linear Mixed Models and Parallel Computing in R

Sydney Benson

University of St. Thomas

Abstract

Generalized linear mixed models are often preferable where non-normal response variables or correlated data are present. Under these conditions, the R package glmm can be used to obtain an appropriate model. Most recently, the abilities of this R package were expanded to decrease the computational expense of model-fitting through parallel computing, executing calculations simultaneously instead of sequentially. glmm uses Monte Carlo likelihood approximation, an iterative importance sampling procedure which estimates the true parameters of a model. Therefore, decreasing the computational expense of the command allows the user to increase their Monte Carlo sample size, giving more accurate estimates without increasing the model-fitting time. Here, generalized linear mixed models are explained, parallel computing is described, and the outcome of combining these two concepts is illustrated through the updates made to the R package glmm.

1 Introduction

1.1 Generalized Linear Mixed Models

Generalized linear mixed models are used, principally, to overcome the limitations of other linear modeling techniques. This approach expands upon the abilities of other techniques by accounting for non-normally distributed and correlated responses. Though generalized linear models can model non-normal responses, they require independence. Similarly, linear mixed models allow for correlated responses, but they assume normality. Thus, generalized linear mixed models advance the techniques used by each of these methods to overcome the assumptions of independence and normality.

The technique that both linear mixed models and generalized linear mixed models use to overcome the independence assumption requires the use of random and fixed effects. Fixed effects are the effect being measured by the model, while random effects are non-observable random variables. Nevertheless, the variance of these random variables can be measured. This means that there are multiple

effects being estimated from the data by the model leading to a more complex likelihood approximation procedure.

The likelihood approximation procedure used for `glmm` [1], created by my research advisor, Dr. Christina Knudson, is Monte Carlo likelihood approximation (MCLA). This procedure uses ordinary Monte Carlo rather than Markov Chain Monte Carlo meaning that the iterative draws used to approximate the likelihood are independent, rather than dependent [2]. To obtain accurate likelihood estimates, many draws must be made, with more draws equating to improved estimates. A moderate number of draws might be 10^5 and takes roughly 43 minutes to be run, illustrating that this approximation procedure is computationally expensive.

1.2 Parallel Computing

The goal of this project was to increase the computational speed of the `glmm` function in the `glmm` package by including a parallel computing component. A parallel computing structure can be used in various programming languages, all needing different syntax. Thus, this project, because it focuses on an `R` package, will detail how a parallel computing structure can be used to run an `R` function in parallel, rather than sequentially. The primary packages that allow us to complete this task are `parallel` and `doParallel`. To summarize, the parallel computing process begins by creating a secondary environment, importing necessary packages and variables from the outside environment, processing the function, naming and exporting the output, and closing the secondary environment.

The parallel computing component is included in the `glmm` package in the objective function. The objective function is used to maximize the approximation of the log-likelihood of the model and must calculate the value of the approximate log-likelihood, the gradient vector, and the Hessian matrix. Since this package uses MCLA to approximate the log-likelihood, using a parallel computing structure to assist with the value, gradient vector and Hessian matrix calculations will decrease the computational expense of the overall likelihood approximation procedure.

To thoroughly discuss the work completed in this package and allow for reproducibility, the other packages utilized as well as the complete parallelizing process will be examined.

2 The Problem

Before the method for the parallelization can be described, the motivator for this project needs to be explained further. Previously, the likelihood approximation procedure used in the package has been described as computationally expensive because the same process of drawing from a distribution is repeated many times. By default, processes in `R` are executed sequentially meaning that the repetitive process used in this package takes a long time to complete, but

there are ways to incorporate parallel computing to allow these processes to be completed simultaneously. Completing processes simultaneously will reduce the computational expense of the likelihood approximation procedure since the draws and subsequent calculations will take only a fraction of the time to process.

To briefly overview the solution to this problem, the parallel computing component within the package is explained. First, the random effects matrix is split row-wise and each available core is utilized to compute the value of the log-likelihood approximation and gradient vector for the respective section of the random effects matrix. This gives a value and gradient vector for each core and so, the values and gradient vectors must be recombined. The recombination of these output is more than simple addition and will be described in more detail later. From these calculations, the vector of weights is also obtained from each core and these vectors are combined to form a single vector. Next, the random effects matrix is split row-wise and the full vector of weights along with the split random effects matrix and the full gradient vector are used to calculate the Hessian matrix. Again, since these calculations are done in parallel, a Hessian matrix is obtained from each core and these separate Hessian matrices need to be recombined. Luckily, since the proper weights and gradient vector are supplied to these calculations, the elements of the Hessian matrices can be added together to form the full Hessian matrix. This process is discussed in greater depth later.

Now that the problem is fully understood and the parallelization process has been outlined, the details of this process can be given.

3 Methodology

3.1 `R` Packages

First, the various packages used to complete the parallelization and the reasoning behind the utilization of each package are explained. Here, the packages `parallel` and `doParallel`, used for general parallelization in `R`, mentioned previously, as well as several other packages used for this specific parallel processing project are explained.

3.1.1 `parallel`

This is one of the most important packages for parallel computing because it allows for the detection of the number of cores in the computing device; creates a computing cluster based on those cores; and allows variables, user-defined functions, and packages to be used within the created computing cluster [3]. Creating a cluster is the beginning of all parallel computing processes because, without a cluster, only one core can be used at a time. The cluster enables the user to access all of the cores within the cluster at the same time so that multiple calculations can be run at once.

Detecting the correct number of cores for the cluster is also a very important process since the number of cores in any computing device can vary and an

optimal number of cores for the computing process needs to be determined. In most cases, the optimal number of cores used for any process will be one less than the number of cores in the machine, since using all cores in a machine can result in restricted use of the device for other activities.

The last important functionality that `parallel` provides is the ability to import variables, user-defined functions and packages from the global environment into the cluster environment for use with the parallel computing calculations. Without this capability, a user would be restricted to creating all necessary variables and functions within their cluster and using only functions already existing in `R`'s base package instead of functions available in other packages or user-defined functions.

3.1.2 `doParallel`

This package is important for most parallel computing processes because it allows the cluster created by the `parallel` package to be used with the `foreach` package, discussed next [4]. By default, without registering the parallel backend, or cluster, prior to use, the `foreach` function will complete loops sequentially rather than simultaneously. Therefore, in order to access `foreach`'s parallel computing capabilities, a parallel backend must be registered using functions in the `doParallel` package.

3.1.3 `foreach`

The next package, `foreach`, allows the `foreach` function to be accessed [5]. This function allows for looping within a cluster as opposed to passing integers to a function in a cluster. The looping capability means that vectors and matrices can be passed to the parallelized function in addition to integers.

3.1.4 `itertools`

The final package used specifically for this parallelization project is `itertools`. This package allows any matrix or vector to be divided into a number of sections [6]. In this project, this package is used to divide the random effects matrix, referred to as the u-matrix in the following sections, into a certain number of sections, by rows, to be sent to the objective function and divide the vector of weights (b-vector) into the same number of sections for calculating the Hessian matrix. This gives the package a way to compute the value of the log-likelihood, gradient and Hessian in parallel.

3.2 The Parallelization Process

The process of parallelizing the objective function was done in two separate steps. First, the calculations for the value of the log-likelihood and gradient vector were completed in parallel. Next, the b-vector and the u-matrix were used to compute the Hessian matrix in parallel. Thus, the process of opening the cluster, creating the proper cluster environment, splitting the u-matrix, and

closing the cluster will be completed twice. The b-vector is only split when calculating the Hessian matrix.

3.2.1 Preparing the Cluster

The very first activity that needs to be completed in any parallel computing process in `R` is preparing the computing cluster. To prepare the cluster, the number of usable cores needs to be determined, the cluster needs to be built and registered, and the proper variables, user-defined functions, and packages need to be imported to the cluster environment. This section will give an overview of this process in more detail, specifying the functions used.

3.2.1.1 `detectCores`

Before anything else, the number of usable cores needs to be determined. This is completed using the function `detectCores`. Since the number of cores used for the parallel computations needs to be one less than the total number of cores available in the device for all devices with more than one core, r is specified so that

$$r = \texttt{detectCores()} - 1$$

However, some machines only have one core. Using the equation above, for these devices, $r = 0$. No calculations can be completed using zero cores, so the core use equation can be altered to be

$$r = \max(\texttt{detectCores()} - 1, 1)$$

This way, if more than one usable core is detected that number of cores will be used, but it is assured that at least one core will always be used.

On the other hand, if the user chooses to specify the number of cores to use, they have that ability. However, the specified number of cores must always be less than or equal to the optimal number of cores determined by the above equation. If a greater number of cores is specified, the function will revert to using the optimal number of cores specified by the equation.

3.2.1.2 `makeCluster`

Once the usable number of cores is determined, the cluster environment can be created. This is done using the `makeCluster` function where the only argument is the number of usable cores to be accessed for the cluster.

3.2.1.3 `registerDoParallel`

Next, register the parallel backend, or cluster environment, for use with the `foreach` function. Again, this is a very important step. Without this step, the `foreach` function would complete calculations on a loop sequentially, similar to the `for` function in `R`. The `registerDoParallel` function allows the `foreach` function to complete calculations on a loop in parallel.

3.2.1.4 `clusterEvalQ`

The `clusterEvalQ` function evaluates an expression on each core within the cluster. `clusterEvalQ` is used in this parallelization to import and install any necessary packages. By using the `library` function within the `clusterEvalQ` function, all necessary packages are installed into the cluster environment. In this case, the `itertools` package is the only package that needs to be used in the cluster environment.

3.2.1.5 `clusterExport`

Finally, the `clusterExport` function is used to finish the cluster preparations. The `clusterExport` function allows any necessary variables and user-defined functions from the global environment to be imported into the cluster environment. We use this function to import all variables needed to run the `C` functions that calculate the value of the log-likelihood approximation and gradient or Hessian, respectively.

3.2.2 Separating the Random Effects Matrix

The next step in the parallel computing process is to separate the u-matrix. This step uses the `isplitRows` function from the `itertools` package. `isplitRows` allows the u-matrix to be split into roughly equal sections based on the number of rows in the matrix. Then, each core within the cluster receives one of these sections.

3.2.3 Separating the Vector of Weights

Finally, for the calculation of the Hessian matrix only, the b-vector must be separated. This is done using the `isplitVector` function, also from the `itertools` package. `isplitVector` splits the b-vector into roughly equal sections based on the number of elements in the vector. Each core in the cluster receives one of these sections and the length of each section coordinates with the number of rows in the section of the u-matrix that the core receives.

3.2.4 Calculating the Value of the Log-Likelihood Approximation and Gradient

To calculate the value of the log-likelihood approximation and gradient, the sections of the u-matrix and other necessary variables are sent to a `.C` function, which calls the `C` function that calculates the value of the log-likelihood approximation and gradient for each section of the u-matrix. The imported variables and u-matrix sections can be sent to their respective cores using the `%dopar%` operator. Then, the values and gradients from each core in the cluster must be combined using the following equations to give the value and gradient for the entire u-matrix. The development of these equations can be found in Appendix A.

3.2.4.1 The Value of the Log-Likelihood Approximation

The output from each cluster is given as a list. One element of this list contains the value of the log-likelihood approximation, referred to as v_i, meaning the value from the ith core. Then, accounting for any computational instability and an uneven division of the rows of the u-matrix among the cores of the cluster, the value of the full u-matrix can be calculated by

$$l_m(\theta|y) = a + \log\left(\frac{1}{m}\sum_{i=1}^{r} j_i \; e^{v_i - a}\right) \tag{1}$$

where $a = \max v_i$, m is the Monte Carlo sample size, or the number of rows of the full u-matrix, and j_i is the number of rows of the section of the u-matrix sent to the ith core.

3.2.4.2 The Gradient

The equation for properly recombining the gradient vectors is slightly more complex than the value of the log-likelihood approximation. Again, the gradient vector is given in the output as one element of a list. This element is referred to as g_i, meaning the gradient from the ith core. Since the gradient is a vector, each element of this vector can be referred to by using $g_{i,c}$ meaning the cth element of the gradient from the ith core. Then, we can calculate the gradient for the full u-matrix using

$$\nabla l_m(\theta|y) = \frac{\sum_{i=1}^{r} j_i \; e^{v_i - a} \; g_{i,c}}{\sum_{i=1}^{r} j_i \; e^{v_i - a}} \tag{2}$$

where $c = 1, 2, \ldots q$ and q is the length of the gradient vector. Like the calculation for the value of the log-likelihood approximation, this calculation also accounts for an uneven division of the rows of the u-matrix among the cores of the cluster and computational instability.

3.2.5 Calculating the Hessian

The Hessian is calculated in a parallel function separate from the value of the log-likelihood approximation and the gradient vector and uses output from the calculation of the two (the b-vector). To calculate the value of the log-likelihood approximation and the gradient, $\frac{f_\theta(u_k, y)}{\tilde{f}(u_k)}$ must be calculated. Hence,

$$\frac{f_\theta(u_k, y)}{\tilde{f}(u_k)} = e^{b_k}$$

if

$$b_k = \log f_\theta(u_k) + \log f_\theta(y|u_k) - \log \tilde{f}(u_k)$$

where k is the kth row of the u-matrix. So, if all b_k's from the output of the first parallel function are combined into one vector, this b-vector can be used in the Hessian calculations. Taking into account computational instability, we can find the Hessian matrix for the full u-matrix using

$$\nabla^2 l_m(\theta|y) = \frac{\sum_{k=1}^{m} \left[\nabla^2 \log f_\theta(u_k, y)\right] e^{b_k - d}}{\sum_{i=1}^{m} e^{b_k - d}} + \frac{\sum_{k=1}^{m} \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right] \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right]^T e^{b_k - d}}{\sum_{i=1}^{m} e^{b_k - d}} \quad (3)$$

where $d = \max(b_k)$.

It is assured that the Hessian is correct by providing $\nabla l_m(\theta|y)$, or the gradient, to the function that calculates the Hessian matrix.

For more information on the development of the Hessian matrix recombination, see Appendix A.

3.2.6 Closing the Cluster

The final step to take in the parallel computing process is to close the cluster. This means that the cluster environment is discarded and all variables which have not been exported in the output of the cluster will be discarded as well. `stopCluster` is used to accomplish this. Once a cluster is closed, the cluster will need to be prepared once again in order to be used.

4 Results

The implementation of the parallel computing component has proved successful for decreasing the computational expense of the `glmm` function for the binomial case. The following example illustrates a case where a generalized linear mixed model could be used due to a non-normal (binomial) response distribution and correlated responses, and how the use of parallel computing has decreased the computational expense of building this model.

In this example, the `salamander` data set, from the `glmm` package, is used. This data set is taken from a biological study that focused on salamander mating preferences. Specifically, they asked whether salamanders preferred to mate with other salamanders from their same location. Additionally, the appearance of the salamander was dependent on their location. The salamanders all came from a single species, but were categorized as rough butt or white side salamanders based on their location of origin.

The response in this data set is correlated because each salamander was mated with multiple other salamanders, leading to repeated measurements from each of the salamanders. It also has a non-normal response distribution because the response is binary, indicating a binomially distributed response. Additionally, each salamander has a personalized tendency to mate and each of those tendencies is independent. Thus, there is a random effects component to the data.

The model can be calculated using the following command in `R`:

```
> sal <- glmm(Mate ~ 0 + Cross,
   random = list( ~ 0 + Female, ~ 0 + Male ),
   varcomps.names = c( "F" , "M" ),
   data = salamander,  m = 10^5,
   family.glmm = bernoulli.glmm)
```

Here, the `0 + Cross` notation is used to obtain log odds of mating estimates for each mating pair type without using a reference group. The `0 + Female/Male` notation is used to center the random effects around 0, a characteristic of most random effects. Finally, `m` denotes the Monte Carlo sample size and the larger `m` is, the more computationally expensive this command becomes, but it provides improved estimates.

Additionally, the run-time for the command has been timed on a MacBook Air, purchased in 2015, containing four cores and 4 GB of RAM, using the `proc.time` command. The processing time for the command with calculations completed sequentially was 2600.622 seconds, or roughly 43 minutes. The processing time for the command using parallel calculations was 2196.815 seconds, or roughly 36 minutes. Thus, by running the calculations in parallel, the user saves approximately 7 minutes.

If a machine with more cores was used, the processing time for the command using parallel calculations would decrease. Additionally, as the Monte Carlo sample size increases, the processing time gap between the command using sequential calculations and the command using parallel calculations will widen.

5 Future Work

Though MCLA is used in the same way to create generalized linear mixed models using a variety of response distributions and the parallel processing component of this package is contained in the calculations leading to MCLA, it would be important to test the parallel calculations using other response distributions. Prior to the changes outlined here with regards to parallel computing, the package could model Poisson distributed responses in addition to the binomial response mentioned earlier. In the future, it would be important to test the parallel computing changes on a data set with a Poisson distributed response.

Finally, unrelated to parallel computing, it is important that this package has the ability to model negative binomially distributed response data in the fu-

ture. This would require developing the canonical link for the negative binomial distribution and incorporating it into the package.

References

[1] Knudson C. (2015). *glmm: Generalized Linear Mixed Models via Monte Carlo Likelihood Approximation.* R package version 1.0.2, URL `http://CRAN.R-project.org/package=glmm`.

[2] Knudson C. (2016). *Monte Carlo Likelihood Approximation for Generalized Linear Mixed Models.* Ph.D. Thesis, University of Minnesota.

[3] Ripley B., Tierney L., Urbanek S. (2017). *Package 'parallel'* R package version 3.3.1, URL `http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`.

[4] Microsoft Corporation and Steve Weston (2017). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package.* R package version 1.0.11, URL `https://CRAN.R-project.org/package=doParallel`.

[5] Microsoft and Steve Weston (2017). *foreach: Provides Foreach Looping Construct for R.* R package version 1.4.4, URL `https://CRAN.R-project.org/package=foreach`.

[6] Steve Weston and Hadley Wickham (2014). *itertools: Iterator Tools.* R package version 0.1.3, URL `https://CRAN.R-project.org/package=itertools`.

A Recombination Equations

A.1 The Value of the Log-Likelihood Approximation

Begin by thinking about combining the values given by each core in the simple case, not accounting for any computational instability or uneven division of the u-matrix among the cores. We can find the value for the entire u-matrix using

$$l_m(\theta|y) = \log\left(\frac{1}{r}\sum_{i=1}^{r} e^{v_i}\right) \tag{4}$$

where r is the number of cores being used and v_i is the value obtained from each core. However, the issue of an uneven distribution of the u-matrix among the cores might arise because the number of rows in the u-matrix is not divisible by the number of cores being utilized. In this case, equation 4 needs to be altered to be

$$l_m(\theta|y) = \log\left(\frac{1}{m}\sum_{i=1}^{r} j_i \; e^{v_i}\right) \tag{5}$$

where m is the number of rows of the u-matrix and j_i is the number of rows of the section of the u-matrix being processed by the ith core. Finally, an overflow problem might occur. This occurs when a number becomes so great that it is stored as infinity by `R`. Thus, the equation changes to

$$l_m(\theta|y) = a + \log\left(\frac{1}{m}\sum_{i=1}^{r} j_i \; e^{v_i - a}\right) \tag{6}$$

where $a = \max v_i$.

A.2 The Gradient Vector

Recombining the gradient from each core requires slightly more work than the recombination of the values. To understand the complexity of this equation, start with the simplest case possible, one with no computational instability and an even distribution of the portions of the u-matrix among the cores. First, the gradient we get from processing the u-matrix on a single core follows this equation:

$$\nabla l_m(\theta|y) = \frac{\sum_{k=1}^{m} \left[\nabla \log f_\theta(u_k, y)\right] \dfrac{f_\theta(u_k, y)}{\tilde{f}(u_k)}}{\sum_{k=1}^{m} \dfrac{f_\theta(u_k, y)}{\tilde{f}(u_k)}} \tag{7}$$

where m is the number of rows in the u matrix. This means that the gradient received from each core via parallel processing will follow this equation. Thus, we face the issue of having an incorrect denominator if we simply add the gradients from each core. To fix this, we need to first multiply the gradient from each core by the denominator of that gradient. We can obtain $\sum_{k=1}^{m} \dfrac{f_\theta(u_k, y)}{\tilde{f}(u_k)}$ since we know that $l_m(\theta|y) = \log\left(\dfrac{1}{m}\sum_{k=1}^{m} \dfrac{f_\theta(u_k, y)}{\tilde{f}(u_k)}\right)$ and $l_m(\theta|y) = \log\left(\dfrac{1}{r}\sum\limits_{i=1}^{r} e^{v_i}\right)$. Thus,

$$\log\left(\frac{1}{m}\sum_{k=1}^{m}\frac{f_\theta(u_k,y)}{\tilde{f}(u_k)}\right)=\log\left(\frac{1}{r}\sum_{i=1}^{r}e^{v_i}\right) \tag{8}$$

$$\frac{1}{m}\sum_{k=1}^{m}\frac{f_\theta(u_k,y)}{\tilde{f}(u_k)}=\frac{1}{r}\sum_{i=1}^{r}e^{v_i} \tag{9}$$

$$\sum_{k=1}^{m}\frac{f_\theta(u_k,y)}{\tilde{f}(u_k)}=\frac{m}{r}\sum_{i=1}^{r}e^{v_i} \tag{10}$$

therefore, we can combine the gradient from each core using

$$\nabla l_m(\theta|y)=\frac{\frac{m}{r}\sum\limits_{i=1}^{r}e^{v_i}\ g_{i,c}}{\frac{m}{r}\sum\limits_{i=1}^{r}e^{v_i}} \tag{11}$$

$$=\frac{\sum\limits_{i=1}^{r}e^{v_i}\ g_{i,c}}{\sum\limits_{i=1}^{r}e^{v_i}} \tag{12}$$

where $g_{i,c}$ is the cth element of the gradient vector produced by the ith core. To alter this equation for handling an uneven distribution of the u-matrix we need to make a change similar to the change made to the value calculation in order to handle this same problem. The new equation is

$$\nabla l_m(\theta|y)=\frac{\sum\limits_{i=1}^{r}j_i\ e^{v_i}\ g_{i,c}}{\sum\limits_{i=1}^{r}j_i\ e^{v_i}} \tag{13}$$

where j_i is the number of rows of the chunk of the u-matrix being processed by the ith core, as mentioned previously. Then, for computational stability, to account for overflow and underflow, we can make a similar alteration to the one made above. Our final equation becomes

$$\nabla l_m(\theta|y)=\frac{\sum\limits_{i=1}^{r}j_i\ e^{v_i-a}\ g_{i,c}}{\sum\limits_{i=1}^{r}j_i\ e^{v_i-a}} \tag{14}$$

and a is defined as above.

A.3 The Hessian Matrix

Finally, consider how to recombine the Hessian. First, the equation to calculate the hessian is

$$\nabla^2 l_m(\theta|y) = \frac{\sum_{k=1}^{m} \left[\nabla^2 \log f_\theta(u_k, y)\right] \frac{f_\theta(u_k, y)}{\tilde{f}(u_k)}}{\sum_{k=1}^{m} \frac{f_\theta(u_k, y)}{\tilde{f}(u_k)}} + \frac{\sum_{k=1}^{m} \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right] \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right]^T \frac{f_\theta(u_k, y)}{\tilde{f}(u_k)}}{\sum_{k=1}^{m} \frac{f_\theta(u_k, y)}{\tilde{f}(u_k)}} \tag{15}$$

Also, remember $\frac{f_\theta(u_k, y)}{\tilde{f}(u_k)} = e^{b_k}$ where $b_k = \log f_\theta(u_k) + \log f_\theta(y|u_k) - \log \tilde{f}(u_k)$. Thus,

$$\nabla^2 l_m(\theta|y) = \frac{\sum_{k=1}^{m} \left[\nabla^2 \log f_\theta(u_k, y)\right] e^{b_k}}{\sum_{i=1}^{m} e^{b_k}} + \frac{\sum_{k=1}^{m} \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right] \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right]^T e^{b_k}}{\sum_{i=1}^{m} e^{b_k}} \tag{16}$$

and so, to account for computational instability, multiply the equation by $\frac{e^d}{e^d}$ where $d = \max(b_k)$

$$\nabla^2 l_m(\theta|y) = \frac{\sum_{k=1}^{m} \left[\nabla^2 \log f_\theta(u_k, y)\right] e^{b_k - d}}{\sum_{i=1}^{m} e^{b_k - d}} + \frac{\sum_{k=1}^{m} \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right] \left[\nabla \log f_\theta(u_k, y) - \nabla l_m(\theta|y)\right]^T e^{b_k - d}}{\sum_{i=1}^{m} e^{b_k - d}} \tag{17}$$

Again, it is assured that the second part of equation 17 is accurate by providing $\nabla l_m(\theta|y)$, the gradient, to the function which calculates the Hessian. Additionally, by collecting the b_k's from each core in the value and gradient calculation, we can assure that the denominator of the equation is the same for each of the cores calculating the Hessian.

Once the elements of the Hessian are collected from each core the final, full hessian for the u-matrix can be calculated. To calculate the final Hessian for the complete u-matrix, sum the elements of the Hessian produced by each core.