Proceedings of the ISCA
22nd International Conference

PARALLEL AND DISTRIBUTED COMPUTING

and COMMUNICATION SYSTEMS

Louisville, Kentucky  USA
September 24-26, 2009

Editors: J. Graham and A. Skjellum

A Publication of
The International Society for
Computers and Their Applications - ISCA

ISBN: 978-1-880843-72-7
A High Performance Gibbs Sampling Algorithm for Item Response Theory

Kyriakos Patsias* Yanyan Sheng† Shahram Rahimi*
*Department of Computer Science, †Department of Educational Psychology & Special Education
Southern Illinois University, Carbondale, IL 62901

Abstract

Item response theory (IRT) is a modern test theory that has been used in various aspects of educational and psychological measurement. The fully Bayesian approach shows promise for estimating IRT models. Given that it is computationally expensive, the procedure is limited in real applications. It is hence important to seek ways to reduce the execution time, and therefore, a suitable solution is the use of high performance computing. This study modifies the existing fully Bayesian algorithm for an IRT model so that it can be implemented on a high performance parallel machine. Empirical results suggest that this parallel version of the algorithm achieves a considerable speedup and thus reduces the execution time considerably.

Keywords: two-parameter IRT model, fully Bayesian estimation, message passing interface, cluster computing

1 Introduction

Item response theory (IRT) provides measurement models that describe a probabilistic relationship between correct responses on a set of test items and a latent trait. With many advantages [1], it has been applied in practically all aspects of educational and psychological measurement, such as test development, item analysis, score reporting, equating, differential item functioning, etc., [1] [2] [3] [4].

1.1 Gibbs sampling for 2PNO IRT models

A major complexity in applying IRT models in various applications lies in the estimation task. Recently, because of the availability of high-computing technology, the attention is focused on fully Bayesian estimation procedures, which provide a number of advantages over the traditional method. Albert [5] applied Gibbs sampling [6], one of the most efficient Markov Chain Monte Carlo (MCMC; e.g., [7]) algorithms, to the two-parameter normal ogive (2PNO; [8]) model, which takes the form

\[ P(y_{ij} = 1) = \Phi(\alpha_j \theta_i - \gamma_j) = \int_{-\infty}^{\alpha_j \theta_i - \gamma_j} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt \]  

for person \( i \) to obtain a correct response for item \( j \), where \( i = 1, \ldots, n \) and \( j = 1, \ldots, k \), \( \alpha_j \) and \( \gamma_j \) denote item parameters and \( \theta_i \) denotes the continuous person trait parameter. The Gibbs sampler involves updating three sets of parameters in each iteration, namely, an augmented \( Z_{ij} \) (which is positive if \( y_{ij} = 1 \) and negative if \( y_{ij} = 0 \)), the person parameter \( \theta_i \), and the item parameters \( \alpha_j \) and \( \gamma_j \). Since a large number of iterations are needed for the Markov chain to reach convergence, the algorithm is computationally intensive and requires considerable amount of execution time, especially with large datasets [9]. Hence, achieving a speedup, and thus reducing the execution time, would make it more practical for researchers or practitioners to implement IRT models using Gibbs sampling.

1.2 High performance computing

High performance computing (HPC) employs supercomputers and computer clusters to tackle problems with complex computations. HPC utilizes the concept of parallel computing to run programs in parallel and achieve a smaller execution time. Many large scale applications and algorithms utilize Message Passing Interface (MPI) standard to achieve better performance. The MPI standard is an application programming interface that abstracts the details of the underlying architecture and network. Some examples of applications that use MPI are crash simulations codes, weather simulation, and computational fluid dynamic codes [10] to name a few.

With the advent of computer clusters, the cost of setting up a HPC system can be relatively low compared to supercomputers. Computer clusters can be set up using PCs or small servers that can be acquired at low cost. Moreover, grid computing technologies make it possible to build a computer cluster which comprises of heterogeneous platforms. This means that existing hardware in an organization

31
may be used to build a cluster comprising of different platforms.

In view of the above, parallel computing can potentially help reduce time for implementing MCMC with the 2PNO IRT model, and as the size of data and/or chain increase, the benefit of using parallel computing would increase. The purpose of this study is hence to develop a high performance Gibbs sampling algorithm for the 2PNO IRT model using parallel computing. The aim is to achieve a high speedup while keeping the cost down. The cost of solving a problem on a parallel system is defined as the product of parallel runtime and the number of processing elements used.

2 Methodology

The study was performed using the Athena Linux cluster, a cluster with nine processing nodes. Athena uses the message passing model via the MPIICH MPI framework implementation. One of the nine nodes acted as the root node, while the rest of the nodes acted as slave nodes. The root node was responsible for generating and partitioning the matrix y, transmitting the submatrices, updating and broadcasting θ, execution time recording, as well as the same duties as the slave nodes.

The root node has an Intel Xeon dual core processor clocked at 3.6 GHz, 4GB of RAM, and Linux 64bit operating system. The eight slave nodes have an Intel Xeon dual core processor clocked at 3.2 GHz, 4 GB of RAM, and Linux 64bit operating system each. MPIICH allows the user to choose how many nodes to use before the execution of a program, thus various number of processing nodes may be used in every execution.

2.1 Parallelism with the Gibbs sampler

When decomposing a problem for parallel computation, the first decomposition method considered is the domain decomposition. In domain decomposition, the data associated with the problem are decomposed and a set of computations are assigned to them [11]. Domain decomposition is a great fit for the 2PNO IRT algorithm since the input and intermediate data can easily be partitioned as illustrated in Figures 1 and 2.

First of all, the root processing node (P₀) partitions the input matrix y, according to the number of available processors, and maps the partitions to the processors as shown in Figure 1.

In this approach, P₁ receives a sub matrix, y₁, of size n × g that corresponds to the elements of the y matrix from y₀,₀ to y₁,₁g−₁, where g = k/P and P is the number of processing nodes. P₁ (second processing node) receives a sub matrix, y₂, of y of size n × g that corresponds to the elements of the y matrix from y₁,₁g to y₂,₁₁−₁g−₁ and so forth. Consequently, each processing node updates the Gibbs samples as in the serial algorithm, but with a smaller input data set. Instead of operating on the whole input matrix y, they operate on a part of it of size n × g.

Decompositions [11] of Z, α, and γ are depicted in Figure 2, where we see that each processor is

![Figure 1: The input y matrix mapped for five processing units.](image1)

![Figure 2: The Z matrix, and α and γ vectors mapped for five processing units.](image2)
updating a block of \( Z, \alpha, \) and \( \gamma. \) For instance, \( P_0 \) is updating a block of \( Z, Z_{P_0}, \) from \( Z_{0,0} \) to \( Z_{N-1,g-1}, \) a block of \( \alpha, \alpha_{P_0}, \) from \( \alpha_0 \) to \( \alpha_{g-1}, \) and a block of \( \gamma, \gamma_{P_0}, \) from \( \gamma_0 \) to \( \gamma_{g-1}. \)

Since \( \theta \) is of size \( n \times 1 \) (a column vector), it is not decomposed. However, a problem arises with the update of \( \theta. \) For simplicity, consider the update of the first element of \( \theta, \) which requires the update of \( \alpha, \gamma, \) and the first row of \( Z. \) Yet, any given processing node has only a part of \( \alpha, \gamma, \) and the first row of \( Z. \) The solution is to assign one of the processing nodes to update \( \theta \) and broadcast it to the rest of the units. The naive approach to update \( \theta \) would be to have all the units send their part of \( \alpha, \gamma \) and \( Z \) to the root so that it has the complete \( Z, \alpha \) and \( \gamma \) to update \( \theta \) and then broadcast \( \theta \) to the rest of the nodes. The problem with this approach is that the data communicated are too large. When the block mapped to every node exceeds one column, i.e. when \( g > 1, \) the communication time is high and the execution time is higher than the serial algorithm.

A better approach is one that minimizes the communication cost. This can be achieved by having every node to calculate \( \psi_i = \sum_j (Z_{ij} + \gamma_j) \alpha_{ij} \) and \( \tau = \sum_j \alpha_j^2, \) and send \( \psi_i \) and \( \tau \) to the root instead. In this way, each processing node is sending a vector of size \( n + 1 \) to the root and one message of size \( n \) is broadcasted by the root. The total data transferred between all the nodes by this approach is

\[
l((n+1) \times P) + l(n \times P) = lP(2n + 1).
\]

The total data transferred between all the nodes by the naive approach is

\[
l((ng + 2g) \times P) + l(n \times P) = lP(g(n+2) + n),
\]

which equals \( lP(2n + 2) \) when \( g = 1, lP(3n + 4) \) when \( g = 2, \) and so forth. When \( g > 1, \) the total data transferred using the naive approach are considerably more than that of the proposed approach (\( n \) is usually in the order of thousands).

2.2 Implementation

The algorithm was implemented in ANSI C and MPI with utilization of the GNU Scientific Library (GSL) [12]. To achieve the parallel computation as illustrated in the previous section, the MPLGather and MPLBcast routines were used for collective communications. See the Appendix for part of the source code of the parallel algorithm in updating the model parameters.

2.3 Performance analyses

In order to investigate the benefits of the proposed parallel solution against its serial counterpart, four experiments were carried out in which sample size (\( n \)), test length (\( k \)), and number of iterations (\( l \)) varied as below:

- \( n = 2000, k = 54, l = 10,000, \)
- \( n = 5000, k = 54, l = 10,000, \)
- \( n = 2000, k = 100, l = 10,000, \)
- \( n = 2000, k = 54, l = 20,000. \)

In all these experiments, one (representing the serial algorithm) to nine processing nodes were used to implement the Gibbs sampler. They were evaluated using four performance metrics in addition to the execution time. These metrics are the total overhead, relative speedup, relative efficiency, and cost:

- The total overhead can be calculated as:
  \[
  T_0 = PT_P - T_S,
  \]
  where \( P \) is the number of available processing nodes, \( T_S \) is the fastest sequential algorithm execution time and \( T_P \) is the parallel algorithm execution time.
- Relative speedup is the factor by which execution time is reduced on \( P \) processors and it is defined as:
  \[
  S = T_S / T_P.
  \]
- Efficiency describes how well the algorithm manages the computational resources. More specifically, it tells us how much time the processors spend executing important computations [11]. Relative efficiency is defined as:
  \[
  E = T_S / PT_P.
  \]
- The definition of cost of solving a problem on a parallel system is the product of parallel runtime and \( P. \) Consequently, cost is a quantity that reveals the sum of individual processing node runtime.

3 Results and Discussion

Results from the four experiments are summarized in Figures 3 to 7. Note that the values plotted represent the average of two runs. As expected, the execution time decreased as the number of processing nodes increased in all the experimented conditions (see Figure 3).
In terms of efficiency and cost, the algorithm performed better using two to five processing nodes (see Figures 4 and 5). When using up to six nodes, the communication overhead (see Figure 6) is sufficiently low in order to not affect the overall speedup (see Figure 7). However, the algorithm had the smallest execution time when nine processing nodes were used (see Figure 3). When nine processing nodes are used, the problem domain is decomposed into even smaller components, allowing every node to perform the computations faster. In this case, the communication overhead effect on the speedup is also lower, and therefore, the overall speedup is greater.

It is noted that the overhead was increased as the number of processing nodes was increased and it reached the maximum with 8 or 9 processing nodes. This is because in the parallel algorithm, the overhead of communication is a result of nodes sending $\psi$ and $\tau$ to the root and then the root broadcasting $\theta$ to the rest of the nodes in every iteration. Note that the total data transferred between all the nodes during execution is $LP(2n + 1)$. The biggest part of idling occurs when the root waits to receive $\psi$ and $\tau$ from all the slave nodes and when the slave nodes wait for the root node to calculate $\theta$ and broadcast it to them. The communication overhead increases more than the computation speedup when a certain amount of processors are used (ranges from four to six processors in the experiments performed). As a result, the speedup does not increase with increasing processor count, and consequently, the cost increases dramatically.

Furthermore, a close examination of Figure 7 indicates that the experiments with input matrix sizes 1000 $\times$ 54, 2000 $\times$ 54, 5000 $\times$ 54, and 2000 $\times$ 54 (with number of iterations $l = 20000$), follow very similar paths. The common parameter of these experiments is the number of items, $k$. The plot for the experiment with input matrix size 2000 $\times$ 100 shows that the algorithm maintains a higher speedup compared to the other experiments. Even though experiments with input matrix size 1000 $\times$ 54 and 2000 $\times$ 54 have smaller input size, the experiment with input matrix 2000 $\times$ 100 maintains a higher speedup over all the processors. The same pattern is observed from Figure 5. In particular, the plot for the experiment with
input matrix size $2000 \times 100$ shows that the algorithm maintains a higher efficiency compared to the other experiments where $k = 54$. These are because the size of the messages communicated in every iteration from the slave nodes to the root, and from the root to the slave nodes, depends only on $n$. As $k$ increases, the message size and communication overhead remain unaffected. Because of this, the processors spend more time performing computations and hence the efficiency and speedup increase.

4 Conclusion

This study developed a high performance Gibbs sampling algorithm for the 2PNO IRT model with the purpose of achieving the lowest execution time possible using the available hardware (Athena cluster). The algorithm was implemented using the ANSI C programming language and the message passing interface. Experiments were performed to evaluate its performance with various parameters.

Results indicated that the parallel algorithm (for the given problem size) performed better, in terms of efficiency and cost, using two to five processing nodes. On the other hand, the algorithm had the smallest execution time when nine processing nodes were used, which was the total number of processing nodes available to this experiment.

The design of a parallel 2PNO IRT model has proved to be justifiable. Given the high data dependencies, such as the dependency of one state of the chain to the previous states, and the dependencies between the data within the same state, the solution initially appeared to be non-trivial. By using domain decomposition, we managed to avoid communication for the state dependencies. Nevertheless, communication in every iteration of the chain cannot be avoided because of the data dependencies within the state. By modifying the serial algorithm, the size of the data communicated in every iteration was managed to be reduced to make a speedup possible.

Further studies can be undertaken to increase the speedup and the efficiency, and minimize the cost and the total overhead. The data may be decomposed differently in order to achieve smaller communication overhead.

Appendix

The pseudo code for updating the values of $Z$, $\psi$, $\tau$, $\theta$, $\alpha$, and $\gamma$ is shown below. Firs of all, $Z$ is updated through the function update.Z. Then, update.PSL.TAU is called to update $\psi$ and $\tau$ and MPI.Gather is called to send $\psi$ and $\tau$ to the root. The root receives $\psi$ and $\tau$ and calls update.TH to update $\theta$ and afterwards broadcasts $\theta$ by calling MPI.Bcast. Finally, $\alpha$ and $\gamma$ are updated from a function call to update.A.G. In order to reduce communication overhead, $\psi$ and $\tau$ are sent in the same message. To achieve that, an array of size $n + 1$ is set up, where the first $n$ entries consist of the elements of $\psi$ and entry $n + 1$ consists of $\tau$ (the name of this array in the code is PSL.TAU.array).

```c
// Start iteration:
for (m = 0; m < l; m++) {
    count++;
    update.Z(Z, y, TH, A, G, r);
    update.PSL.TAU(PSL.TAU.array, Z, A, G);
    MPI.Gather(PSL.TAU.array, n+1, MPI.DOUBLE,
               PSL.TAU.rec, n+1, MPI.DOUBLE, ROOT,
               MPI.COMM_WORLD);
    if (rank == ROOT) {
        double TAU.array[size];
        int ind = 0;
        // Retrieve PSI and TAU from PSL.TAU_rec:
        for (i = 0; i < size; i++) {
            for (j = 0; j < n; j++) {
                if (i < n)
                    gsl_matrix_set(PSL.matrix, i, j,
                                   PSL.TAU.rec[ind++]);
                else
                    TAU.array[i] = PSL.TAU.rec[ind++];
            }
        }
        update.TH(TH, THV, TAU.array, PSL.matrix,
                  count, r);
        // Transfer TH data into a buffer so that it can be broadcasted:
        for (i = 0; i < l; i++) {
            TH.array[i] = gsl_vector_get(TH, i);
        }
        MPI.Bcast(TH.array, n, MPI.DOUBLE, ROOT,
                   MPI.COMM_WORLD);
    }
}```
// Transfer TH received to a vector structure:
for(i=0; i < n; i++)
{
    gsl_vector_set(TH, i, TH_array[i]);
}
update_A,G(A, G, AV, GV, Z, TH, unif, count, r, p);
} // end iteration

References


