1 Introduction

Item response theory (IRT)\cite{1} gives measurement models that describe a probabilistic relationship between correct responses on a set of test items and a latent trait. Such models find applications in education and psychology as well as in other fields. Simultaneous parameter estimation in IRT models leads to statistical complexities which have made the estimation process a major focus of psychometric research.

Fully Bayesian estimation procedures offer a number of advantages compared to other methods \cite{2}, however they are computationally expensive. Albert \cite{3} applied Gibbs sampling, an efficient Markov Chain Monte Carlo (MCMC) algorithm to the two-parameter normal ogive (2PNO) model. However, a large number of iteration is needed for the Markov chain to converge and as a result the algorithm is computationally intensive and demands significant execution time.

The latter fact motivates the application of parallel computing to help reduce time for implementing MCMC with the 2PNO IRT model. There have been a couple of attempts \cite{4, 5} to efficiently parallelize the Gibbs sampling for the 2PRNO IRT model, however they do not scale sufficiently. In this paper, we propose a new parallel algorithm with a novel domain decomposition that theoretically has better scalability properties. In practice, we observe that our algorithm outperforms significantly the state-of-the-art parallel algorithm, achieving a speedup up to 9.7× over it.

2 Background and related work

2.1 The IRT model and Gibbs Sampling

In this study we are focusing on the 2PNO IRT model, which models the person-item interaction by assuming one ability dimension. Let’s assume that a test consists of $k$ multiple-choice items and each one measures a single unified ability $\theta$. Let $y = [y_{ij}]_{n \times k}$ represent a matrix of $n$ examinees’ responses to $k$ items so that $y_{ij}$ is defined as:

$$y_{ij} = \begin{cases} 
1 & \text{if person } i \text{ answers item } j \text{ correctly} \\
0 & \text{if person } i \text{ answers item } j \text{ incorrectly} 
\end{cases}, \ i = 1, \ldots, n \text{ and } j = 1, \ldots, k$$

The probability of person $i$ obtaining correct response for item $j$ can be defined as

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

(1)

where $a_j$ and $\gamma_j$ denote item parameters, $\theta_i$ denotes the continuous person trait parameter and $\Phi$ denotes the unit normal cdf.

The Gibbs sampler involves updating three sets of parameters in each iteration, namely, an augmented continuous variable $Z_{ij}$ (which is positive if $y_{ij} = 1$ and negative if $y_{ij} = 0$), the person parameter $\theta_i$ and the item parameters $\xi_j$, where $\xi_j = (a_j, \gamma_j)$ from their respective full conditional distributions:

$$Z_{ij} \sim \begin{cases} 
N(0, \infty)(a_j \theta_i - \gamma_j) & \text{if } y_{ij} = 1 \\
N(-\infty, 0)(a_j \theta_i - \gamma_j) & \text{if } y_{ij} = 0 
\end{cases}$$

(2)
\[ \theta_i \sim \mathcal{N}\left( \frac{\sum_j (Z_{ij} + \gamma_j) a_j + \mu}{1/\sigma^2 + \sum_j a_j^2}, \frac{1}{1/\sigma^2 + \sum_j a_j^2} \right) \]  

(3)

\[ \xi_j \sim \mathcal{N}\left( (x'x)^{-1} x'Z_j, (x'x)^{-1} I(a_j > 0) \right) \]  

(4)

where \( x = [\theta, -1] \), \( a_j > 0 \) and \( p(\gamma_j) \propto 1 \).

So, with starting values \( \theta^{(0)} \) and \( \xi^{(0)} \), observations \( (Z^{(l)}, \theta^{(l)}, \xi^{(l)}) \) can be simulated from the Gibbs sampler by iteratively drawing from their respective full conditional distributions specified in equations 2, 3 and 4. Hence, to go from \( (Z^{(l-1)}, \theta^{(l-1)}, \xi^{(l-1)}) \) to \( (Z^{(l)}, \theta^{(l)}, \xi^{(l)}) \) we follow three steps:

1. Draw \( Z^{(l)} \sim p(Z | \theta^{(l-1)}, \xi^{(l-1)}) \)
2. Draw \( \theta^{(l)} \sim p(\theta | Z^{(l)}, \xi^{(l-1)}) \)
3. Draw \( \xi^{(l)} \sim p(\xi | Z^{(l)}, \theta^{(l)}) \)

This iterative procedure yields a sequence of \( \{(\theta^{(l)}, \xi^{(l)}), l = 0, \ldots, L \} \). To reduce the effect of the starting values, early iterations in the markov chain are set as burn-ins to be discarded. Finally, samples from the remaining iterations are used to summarize posterior density of item parameters \( \xi \) and ability parameters \( \theta \).

Since we may have to perform thousands of iterations, a single execution of the algorithm can be computationally expensive. For example, [6] showed that for a matrix \( y_{2000 \times 10} \) and 10,000 iterations the algorithm may take 13 minutes when implemented in Fortran using the Microsoft PoweStation 4.0 compiler and the IMSL Fortran library. For a longer chain with 50,000 iterations it takes 60-90 minutes for each execution. This fact further limits the use of IRT models under fully Bayesian framework. However, by leveraging parallel processing we could potentially decrease the execution time and make the algorithm practical in various applications.

2.2 Parallel processing and previous parallelization approaches

As explained in the previous section, the algorithm we are studying is computationally expensive and this motivates the development of parallel algorithms. The cost of a parallel algorithm consists of two parts: computation and communication. Basically, if there are data-dependencies among the parallel computations, the processors have to communicate and this induces the communication cost of the algorithm. However, this communication overhead can significantly affect the parallel performance. Ultimately, in case that the communication cost is significant, the speedup over the serial execution can be minor or, even worse, parallel processing may even slow down the execution. Consequently, we have to design parallel algorithms that minimize communication.

The communication cost can be further divided in two parts:

- **Bandwidth**, which is the number of words moved between processors along the critical path
- **Latency**, which is the number of messages sent along the critical path

Under the alpha-beta model, where \( a \) is the cost of sending a message and \( \beta \) is the time needed to inject a single word into the communication network, an algorithm that moves \( BW \) words and sends \( L \) messages along the critical path spends \( aL + \beta BW \) time units for communication.

There have been a couple of attempts to parallelize the Gibbs sampling for IRT models. First, Patsias et al [4] used a column decomposition of the input matrix \( y \). As it can be easily concluded from equation 3, dependencies arise when we have to compute \( \theta_i \) and these dependencies result in communication among processors. In this particular algorithm, we have to move \( (2n + 1) \log p \) words along the critical path (bandwidth cost) for each iteration. Moreover, we have to send \( 2 \log p \) messages along the critical path (latency cost).

Sheng [5] makes the observation that \( k \ll n \) in practice and uses a row decomposition of the matrix \( y \). Now, there are dependencies among processors regarding the computation of \( \xi_j \) (see equation 4). This algorithm has a bandwidth cost of \( (4k + 2) \log p \) words and a latency cost of \( 2 \log p \) messages. Since \( k \ll n \) there is a big saving in bandwidth and actually this algorithm scales better than the previous one.
In the next section we present a novel algorithm with a 2D domain decomposition that provably minimizes further the communication costs and thus offers more scalability opportunities.

3 A new parallel algorithm using 2D decomposition

3.1 2D domain decomposition

In our parallelization approach we will follow 2D domain decomposition. Essentially we have to partition the matrix $y$ as well as vectors $\theta$ and $\xi$. Let’s assume that we have $p$ processors available for the execution. We arrange them in a $\sqrt{p} \times \sqrt{p}$ square grid and let’s enumerate them in a row-wise fashion. Then we partition the input matrix $y_{n \times k}$ into $(n/\sqrt{p}) \times (k/\sqrt{p})$ blocks and assign such a block to a particular processor as depicted in Figure 1. Regarding vectors $\theta$ and $\xi$ we partition them in $n/\sqrt{p} \times 1$ and $k/\sqrt{p} \times 1$ blocks respectively. Then we replicate the first block of $\theta$ to the first row of processors, the second block of $\theta$ to the second row of processors and so on. Similarly, we replicate the first block of $\xi$ to the first column of processors, the second block of $\xi$ to the second column of processors and so on. We note here that we replicate the blocks of $\theta$ and $\xi$ and this fact is key into achieving an efficient 2D decomposition as it will be highlighted in the next section.

3.2 The new parallel algorithm

By following the domain decomposition described in the previous section the parallel algorithm proceeds as follows: First, each processor computes the quantities $Z_{ij}$ that correspond to the block of $y$ assigned to it by using the formula:

$$Z_{ij} \sim \begin{cases} \mathcal{N}_{(0, \infty)}(a_j \theta_i - \gamma_j), & \text{if } y_{ij} = 1 \\ \mathcal{N}_{(-\infty, 0)}(a_j \theta_i - \gamma_j), & \text{if } y_{ij} = 0 \end{cases}$$

We note here that in this first stage of the algorithm all the available processors work in parallel since they have all the required values of $y_{ij}$, $a_j$, $\theta_i$ and $\gamma_j$. In the second step of the algorithm, we have to draw samples for $\theta_i$ by using equation 3. We observe that for each $\theta_i$ we need the values of $Z_{ij}$ and $\xi_j = (a_j, \gamma_j)$ for $j = 1, \cdots, k$ and each processor has just a part of these needed values. So, processors in the same row

Figure 1: 2D domain decomposition where $y$ is a $30 \times 15$ matrix and there are 9 available processors. The processors $p_0 \cdots p_8$ are arranged into a $3 \times 3$ grid. The input matrix $y_{30 \times 15}$ is partitioned in $10 \times 5$ blocks and each block is assigned to a specific processor. Vector $\theta$ is partitioned into $10 \times 1$ blocks. The first block of $\theta$ is replicated to processors $p_0, p_1, p_2$ (first row of processors), the second block of $\theta$ is replicated to processors $p_3, p_4, p_5$ and the last block of $\theta$ is replicated to processors $p_6, p_7, p_8$. Similarly, vector $\xi$ is partitioned into $5 \times 1$ blocks. The first block of $\xi$ is replicated to processors $p_0, p_3, p_6$ (first column of processors), the second block of $\xi$ is replicated to processors $p_1, p_4, p_7$ and the last block of $\xi$ is replicated to processors $p_2, p_5, p_8$. 

In the second step of the algorithm, we have to draw samples for $\theta_i$ by using equation 3. We observe that for each $\theta_i$ we need the values of $Z_{ij}$ and $\xi_j = (a_j, \gamma_j)$ for $j = 1, \cdots, k$ and each processor has just a part of these needed values. So, processors in the same row
have to cooperate in order to calculate the corresponding \( \theta_i \). More specifically, each processor computes independently the quantities

\[
\psi_i = \sum_{j=1}^{k/\sqrt{p}} (Z_{ij} + \gamma_j) a_j \quad \text{for } i = 1, \ldots, n/\sqrt{p}
\]

(5)

\[
\tau = \sum_{j=1}^{k/\sqrt{p}} a_j^2
\]

(6)

that correspond to their blocks of \( Z, a \) and \( \gamma \). Then, we do sum-reduction operations of the aforementioned quantities along rows of processors as illustrated in Figure 2b. After these reductions, each processor in the first row has obtained terms of the form \( \sum_{i=1}^{n/\sqrt{p}} \psi_i \) and \( \sum_{i=1}^{n/\sqrt{p}} \tau \) and therefore it can draw samples for the corresponding \( \theta_i \)s by using the equation:

\[
\theta_i \sim \mathcal{N}\left( \frac{\sum_{i=1}^{n/\sqrt{p}} \psi_i + \mu}{1/\sigma^2 + \sum_{i=1}^{n/\sqrt{p}} \tau}, \frac{1}{1/\sigma^2 + \sum_{i=1}^{n/\sqrt{p}} \tau} \right) \quad \text{for } i = 1, \ldots, n/\sqrt{p}
\]

(7)

After this step, the processors in the last column should broadcast the newly-drawn samples of \( \theta_i \)s back to the processors in the same row (Figure 2b), so that each one has the appropriate new \( \theta_i \)s required for the next step of the algorithm. At this final step, we have to draw samples for \( \xi_j = (a_j, \gamma_j) \) with \( j = 1, \ldots, k \) by using equation 4. So, each processor computes independently the quantities:

\[
x'x = \begin{bmatrix}
\theta_0 \\
-1 \\
\vdots \\
\theta_{n/\sqrt{p}} \\
-1
\end{bmatrix} \begin{bmatrix}
\theta_0 & -1 \\
\theta_1 & -1 \\
\vdots & \vdots \\
\theta_{n/\sqrt{p}} & -1
\end{bmatrix} = \begin{bmatrix}
\sum_{i=1}^{n/\sqrt{p}} \theta_i^2 \\
-\sum_{i=1}^{n/\sqrt{p}} \theta_i n/\sqrt{p}
\end{bmatrix}
\]

(8)

\[
x'Z_j = \begin{bmatrix}
\theta_0 \\
-1 \\
\vdots \\
\theta_{n/\sqrt{p}} \\
-1
\end{bmatrix} \begin{bmatrix}
Z_{0j} \\
\vdots \\
Z_{n/\sqrt{p}j}
\end{bmatrix} = \begin{bmatrix}
\sum_{i=1}^{n/\sqrt{p}} \theta_i Z_{ij} \\
-\sum_{i=1}^{n/\sqrt{p}} Z_{ij}
\end{bmatrix}
\]

(9)

that correspond to their portions of \( \theta \) and \( Z \). Then, we do sum-reduction operations of these two quantities along columns of processors as illustrated in Figure 2c. After these reductions, each processor in the first row has obtained terms of the form \( \sum_{i=1}^{\sqrt{p}} x'x \) and \( \sum_{i=1}^{\sqrt{p}} x'Z_j \) and thus it can draw samples for the corresponding \( \xi_j \)s by using the equation:

\[
\xi_j \sim \mathcal{N}\left( \left( \sum_{i=1}^{\sqrt{p}} x'x \right)^{-1} \left( \sum_{i=1}^{\sqrt{p}} x'Z_j \right) \right) I(a_j > 0) \quad \text{for } j = 1, \ldots, k/\sqrt{p}
\]

(10)

Finally, the processors in the first row should broadcast the newly-drawn samples of \( \xi_j \)s back to the processors in the same column (Figure 2d), so that each one has the appropriate new \( \xi_j = (a_j, \gamma_j) \)s required for the next iteration of the Gibbs sampling procedure.

### 3.3 Communication costs of the new algorithm

In this section we will derive the communication costs of our new algorithm, since they will determine its scalability. We assume that the collective communication operations involved (sum-reduction and broadcast) exploit the binomial tree algorithm [7]. In this model, a sum-reduction or a broadcast of \( W \) words among \( P \) processors has bandwidth cost of \( W \log P \) words and latency cost of \( \log P \) messages.

So, during the first row sum-reduction each processor contributes a vector \( n/\sqrt{p} \times 1 \) of \( \psi_i \)s and a word for \( \tau \). Also, this operation involves \( \sqrt{p} \) processors and therefore it costs:

\[
\text{Bandwidth}_1: \left( \frac{n}{\sqrt{p}} + 1 \right) \log \sqrt{p} = \left( \frac{n}{2\sqrt{p}} + \frac{1}{2} \right) \log p
\]

(11)
(a) Row sum-reductions of quantities $\psi_i$ and $\tau$ (equations 5 and 6). After the reductions, processors $p_2$, $p_5$ and $p_8$ sample $\theta_i$s.

(b) Processors $p_2$, $p_5$ and $p_8$ broadcast $\theta_i$s back to the processors in the same row since they are needed at the next step.

(c) Column sum-reductions of quantities $x'x$ and $x'Z_j$ (equations 8 and 9). Then, processors $p_0$, $p_1$ and $p_2$ sample $\xi_j = (a_j, \gamma_j)$.

(d) Processors $p_0$, $p_1$ and $p_2$ broadcast $\xi_j$s back to the processors in the same column since they are needed at the next iteration.

Figure 2: Illustration of communication operations in the parallel algorithm using 2D decomposition
Latency: \[ \log \sqrt{p} = \frac{1}{2} \log p \] (12)

During the row-broadcast, the roots send a vector \( n/\sqrt{p} \times 1 \) hence the costs are:

\[ \text{Bandwidth}_2: \frac{n}{\sqrt{p}} \log \sqrt{p} = \frac{n}{2\sqrt{p}} \log p \] (13)

\[ \text{Latency}_2: \log \sqrt{p} = \frac{1}{2} \log p \] (14)

During the column sum-reduction each processor contributes a vector \( 2 \times k/\sqrt{p} \times x'_{Z_j} \) and two words for \( x'x \) (note that due to its special form, only two words are needed to reconstruct \( x'x \)). Also, this operation involves \( \sqrt{p} \) processors and therefore it costs:

\[ \text{Bandwidth}_3: \left( \frac{2k}{\sqrt{p}} + 2 \right) \log \sqrt{p} = \left( \frac{k}{\sqrt{p}} + 1 \right) \log p \] (15)

\[ \text{Latency}_3: \log \sqrt{p} = \frac{1}{2} \log p \] (16)

During the column-broadcast, the roots send a vector \( 2 \times k/\sqrt{p} \) since \( \xi_j = (a_j, \gamma_j) \). Therefore the costs are:

\[ \text{Bandwidth}_4: \frac{2k}{\sqrt{p}} \log \sqrt{p} = \frac{k}{\sqrt{p}} \log p \] (17)

\[ \text{Latency}_4: \log \sqrt{p} = \frac{1}{2} \log p \] (18)

So, if we sum equations 11, 13, 15 and 17 the total bandwidth cost is:

\[ \text{Bandwidth}_{2D}: \frac{n}{\sqrt{p}} \log p + \frac{2k}{\sqrt{p}} \log p + \frac{3}{2} \log p \] (19)

By summing equations 12, 14, 16 and 18 the total latency cost is:

\[ \text{Latency}_{2D}: 2 \log p \] (20)

Let’s rewrite the communication costs for the state-of-the-art algorithm that uses row-wise decomposition:

\[ \text{Bandwidth}_{row}: (4k + 2) \log p = 4k \log p + 2 \log p \] (21)

\[ \text{Latency}_{row}: 2 \log p \] (22)

We can easily conclude from equations 20 and 22 that both algorithms have the same latency cost. In respect to bandwidth, we can not draw a conclusion with the existing forms. However, since \( n \gg k \) we can write \( n = ak \) with \( a \gg 1 \) and then equation 19 becomes:

\[ \text{Bandwidth}_{2D}: \frac{ak}{\sqrt{p}} \log p + \frac{2k}{\sqrt{p}} \log p + \frac{3}{2} \log p = \frac{(a+2)k}{\sqrt{p}} \log p + \frac{3}{2} \log p \] (23)

Now, by comparing equations 21 and 23 we can easily see that if \( p > \left( \frac{a+2}{4} \right)^2 \) then:

\[ \text{Bandwidth}_{2D} < \text{Bandwidth}_{row} \]

Essentially, our new algorithm communicates less words compared to the state-of-the-art algorithm as \( p \) increases and thus we expect to observe better scalability.
4 Performance results

The algorithm was implemented in C and MPI with utilization of the GNU Scientific Library. All the experiments were executed on Hopper. Hopper is a 6,384 node Cray XE-6 machine located at NERSC. Its currently ranked #19 on the Top500 list. Each Hopper node has two 12-core, AMD MagnyCours processors running at 2.1 GHz, yielding 24 cores per node and 153,216 cores in total. Nodes are connected in a three-dimensional torus via the Cray Gemini interconnect.

Here we are focusing on strong scaling experiments. In a strong scaling experiment, we fix the problem size and we increase the number of processors $p$. By doing so, ideally we would expect the execution time to decrease proportionally to the participating number of processors. However, since the communication overhead is increased as more processors are involved, the total execution time does not decrease linearly. The metric we are going to use is called parallel efficiency. Parallel efficiency is defined as:

$$\text{parallel efficiency} = \frac{t_{\text{serial}}}{p \cdot t_{\text{parallel}}}$$

where $t_{\text{serial}}$ is the execution time of the algorithm on a single processor and $t_{\text{parallel}}$ is the execution time of the parallel algorithm using $p$ processors. Thus, the ideal parallel efficiency is 1 for any number of processors.

Figure 3 shows strong scaling results for a problem where $n = 2048$ and $k = 128$. Also, the number of processors varies from 4 to 256 in this case. There are a few things to observe here. First we can see that our algorithm (green line) scales much better than the state-of-the-art algorithm (red line) as expected in theory. For example, at the scale of 256 processors our algorithm is $2.28 \times$ faster. Also, in this particular example our algorithm becomes faster than the one using row-wise decomposition when using 64 processors or more. This fact can be explained by theory. Recall that our algorithm communicates less words if if $p > \left(\frac{a+2}{4}\right)^2$. In this case, $n = 2048 = 16 \times 128 = 16 \times k$ so $a$ is 16. Therefore, our algorithm should be faster if $p > 21$, which is validated by the data.

Figure 4 illustrates the time breakdown for one iteration of this particular problem instance. As we can see, for 64 and 256 processors our algorithm communicates much less. More specifically, for 256 processors our algorithm spends $3 \times$ less time for communication. We also observe that the computation time is slightly decreased comparing to the algorithm using row-wise decomposition. Again we can predict this behavior in theory. Let’s assume that it takes on average $t_{\text{norm}}$ time units to draw a sample from a normal distribution.
Moreover, both sampling $\theta_i$s and $\xi_j = (a_j, \gamma_j)$s lie on the critical path of the parallel algorithms. In the case of the algorithm using row-wise decomposition, $\theta_i$s are sampled in parallel using $p$ processors while $\xi_j$s are sampled by a single processor. Thus, sampling $\theta_i$s and $\xi_j$s takes:

$$t_{row} = \left( \frac{n}{p} + 2k \right) t_{norm} = \left( \frac{ak}{p} + 2k \right) t_{norm}$$

(24)

On the other hand, our algorithm using 2D decomposition uses $\sqrt{p}$ processors for sampling $\theta_i$s and $\sqrt{p}$ processors for sampling $\xi_j$s. Hence, sampling $\theta_i$s and $\xi_j$s takes:

$$t_{2D} = \left( \frac{n}{\sqrt{p}} + \frac{2k}{\sqrt{p}} \right) t_{norm} = \left( \frac{ak}{\sqrt{p}} + \frac{2k}{\sqrt{p}} \right) t_{norm} = \left( \frac{(a + 2)k}{\sqrt{p}} \right) t_{norm}$$

(25)

So, from equations 24 and 25 we can easily conclude that as $p$ increases we expect the 2D algorithm to spend less time for computation on the critical path.

Finally, in Figure 5 we exhibit a strong scaling experiment with a larger problem size ($n = 10240$ and $k = 512$). In this problem instance, the number of processors varies from 4 to 1024. Again we can see that our algorithm using a 2D decomposition scales much better than the algorithm using row decomposition. Specifically, since $n = 10240 = 20 \times 512 = 20 \times k$ we would expect our algorithm to be faster for $p > 31$, which is validated by the data. At the scale of 1024 processors our algorithm is $9.7 \times$ faster comparing to the state-of-the-art parallel algorithm. Also, since the problem is sufficiently large our algorithm runs with 0.83 parallel efficiency at the scale of 1024 processors, meaning that we achieve a $850 \times$ speedup over the serial algorithm. This latter fact is of major importance since makes the computationally intensive algorithm practical in various applications. Another thing worth-mentioning here is the behavior of the parallel algorithms for $p = 16$ and $p = 64$ where the parallel efficiency is larger than 1. This phenomenon is quite often in parallel processing and is called super-linear speedup. Since the working set is quite large, it doesn’t fit in cache (fast memory) of a single processor, while the working sets in the parallel algorithms are smaller and so they fit in cache. As a result, the computation part of the algorithm is decreased super-linearly, yielding ultimately a super-linear speedup.
5 Conclusions

In this paper we have presented a high performance parallel Gibbs sampler for IRT models that minimizes the communication compared to the state-of-the-art parallel algorithm. In doing so, we deployed a novel 2D domain decomposition and we used replication to minimize dependencies. We analyzed the communication costs and we found that given a particular problem instance, there is a point in terms of processor count, after which our algorithm communicates less words and thus it has better scalability properties. Also, we found that our new algorithm employs more parallelism compared to the state-of-the-art algorithm. Finally we presented experimental results on a thousand of processors for a Cray XE-6 machine which show that the theoretical results are validated in practice and our algorithm is up to $9.7 \times$ faster compared to the state-of-the-art parallel algorithm. Since our novel parallel algorithm is quite scalable, we used 1024 processors and we achieved a $850 \times$ speedup over the serial execution for a large problem instance, making the computationally intensive algorithm practical.

References


