PARALLEL GAUSS-SEIDEL ON A TORUS NOC ARCHITECTURE

*Khaled Day and **Mohammad H. Al-Towaiq
*Sultan Qaboos University
**Jordan University of Science and Technology
*P.O. Box 36, Al-Khod 123, Oman
**B. O. Box 3030, Irbid 22110, Jordan
* kday@squ.edu.om, ** towaiq@just.edu jo

ABSTRACT
We propose a parallel Gauss-Seidel (GS) iterative algorithm for solving large systems of linear equations on a 2D torus network-on-chip (NoC) architecture. The proposed parallel algorithm is of $O(Nn^2/k^2)$ time complexity for solving a system with matrix of order $n$ on a $k \times k$ torus NoC architecture requiring $N$ iterations assuming $n$ and $N$ are large compared to $k$ (i.e. for large linear systems that require a large number of iterations). We show that under these conditions the proposed parallel GS algorithm has near optimal speedup and efficiency.

KEY WORDS
Network-on-Chip; 2D Torus, Parallel Processing; Linear System of Equations; Gauss-Seidel

1. Introduction
Solving a system of linear equations of the form:

$$Ax = b$$  \hspace{1cm} (1)

where $A$ is an $n$ by $n$ dense matrix, $b$ is a known $n$-vector and $x$ is an $n$-vector to be determined, is one of the most important classes of problems. There are two classes of methods for solving such linear systems of equations, direct and iterative methods. A direct method involves a fixed number of operations carried out once, at the end of which the solution is produced. Direct methods are often too expensive in terms of computation time. As an alternative, linear systems are usually solved using iterative methods. A method is called iterative when it consists of a basic series of operations which are carried out over and over again, until the answer is produced, some exception error occurs, or a limit on the number of iterations is exceeded [1]. The Gauss-Seidel is one of the most efficient iterative methods for solving linear systems that arise in solving partial differential equations. It is amenable for parallel computing. Many parallel implementations have been proposed including those reported in [2], [3], [4], [5] and [6].

The time spent on communication between processors in a parallel computation limits its speedup and efficiency. With advances in technology, chips with a large number of processing elements (cores) have become a reality. Communication between processing elements in such multicore systems was initially based on buses. When the number of cores increased, the bus became a performance bottleneck. In recent years, networks-on-chip (NoCs) have been used instead of buses for interconnecting the on-chip processing elements which has resulted in faster inter-processor communication. The topology of the network-on-chip has a major impact on the communication performance of the multi-core system [7]. Several topologies have been proposed for NoCs including mesh-based and tree-based topologies [7].

Motivated by these new developments in technology and the resulting improved performance of inter-processor communication on modern network-on-chip systems, we propose and analyze the complexity of a new parallel Gauss-Seidel algorithm designed to run on torus network-on-chip architectures. We show that the proposed parallel algorithm is nearly optimal for large linear systems that require a large number of iterations.

2. Preliminaries
The Gauss-Seidel (GS) algorithm corrects the $i^{th}$ element of the vector $x^{(m)}$, $x_j^{(m)}$ in the order $i = 0, 1, \ldots, n-1$. The approximate solution is updated immediately after the new component is determined. The newly computed element $x_j^{(m+1)}$ can be changed within a working vector which is redefined at each step. This results in the iterative formula (2) and the corresponding sequential GS algorithm outlined in Figure 1.

$$x_j^{(m+1)} = (b_j - \sum_{j=0}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{n-1} a_{ij} x_j^{(m)}) / a_{ii} \hspace{1cm} (2)$$

It is well known that the GS algorithm will always converge if the matrix $A$ is strictly or irreducible diagonally dominant. The sequential GS algorithm has time complexity $O(Nn^2)$ for $N$ iterations and therefore requires large execution time for large problem sizes ($n$)
especially when a large number of iterations $N$ is required, hence the need for faster parallel implementations.

```
Sequential_GS
{}
//Input A, b, x, tolerance
for m = 0, 1, ..., N
{}
    for i = 0, ..., n-1
        sum = 0
        for j = 0, 1, ..., i-1
            sum = sum + $a_{ij}x_{j}^{(m+1)}$
        for j = i+1, ..., n-1
            sum = sum + $a_{ij}x_{j}^{(m)}$
        $x_{i}^{(m+1)} = (b_{i} - \text{sum}) / a_{ii}$
{}
If $(\|x^{(m+1)} - x^{(m)}\| < \text{tolerance})$ then
    output the solution and stop
    update $x^{(m)}$
{}
```

Figure 1: Sequential Gauss-Seidel algorithm

A 2D mesh Network-on-Chip (NoC) consists of $k \times k$ switches interconnecting $k^2$ nodes (processing elements) as illustrated in Figure 2. One disadvantage of the 2D mesh topology is its long diameter which has a negative effect on the communication latency.

![Figure 2: A 2D Mesh NoC Architecture ($k = 4$)](image)

A torus NoC (figure 3a) is the same as a 2D mesh NoC with the exception that the switches on the edges are connected to the switches on the opposite edges with wrap-around links. Every switch in a torus has five active ports: one connected to the local node, and the other four connected to the four neighboring switches (left, right, up, and down). The torus topology reduces the latency of the 2D mesh while keeping its simplicity. In order to reduce the length of the wrap-around links, the torus can be folded as shown in figure 3b.

![Figure 3: (a) Torus and (b) Folded Torus NoC Architectures ($k = 4$)](image)

Mesh-based topologies (especially the 2D mesh and the torus) have been the most popular among the known NoC topologies due to their modularity, their ability to be partitioned into smaller meshes, their simple routing strategy, and their facilitated implementation. They also have a regular structure and short inter-switch wires. They have been used in many systems such as the RAW processor [8], the TRIPS processor [9], the 80-node Intel’s Teraflops research chip [10], and the 64-node chip multiprocessor from Tilera [11].

3. The Proposed Parallel GS Algorithm

In the proposed parallel GS algorithm we partition the $n \times n$ matrix $A$ into blocks of $nk^2$ columns each and scatter them to the $k^2$ processors of the $k \times k$ torus. Each processor is identified by its $(x, y)$ coordinates, $0 \leq x, y \leq k-1$, as illustrated in figure 3. Processor $(x, y)$ is connected to the
four neighboring processors (x-1, y), (x+1, y), (x, y-1) and (x, y+1). Notice that in these expressions the (+1) and (-1) operations on the coordinates are modulo k in order to cover for the wrap-around links. We also assign sequential processor ids to the $k^2$ processors as follows: the processor whose coordinates are (x, y) is assigned the sequential id: $x + ky$. In this way, the $k^2$ processors are identified with sequential ids in the range $0 ... k^2-1$.

Parallel GS Algorithm // Input A, b, x, tolerance

Let $r$ be the local processor sequential id
if ($r = 0$)
    scatter the blocks of columns to the processors
    scatter x to the processors
    // P_r gets the $x_i$'s for $r(n/k^2) \leq j < (r+1)(n/k^2)$
    broadcast tolerance to all processors
else
    receive the $r^{th}$ block of A's columns
    // receive $a_{ij}$'s for $0 \leq i < n$, $r(n/k^2) \leq j < (r+1)(n/k^2)$
    receive the $r^{th}$ segment of the x vector
    // receive $x_i$'s for $r(n/k^2) \leq j < (r+1)(n/k^2)$
    receive the broadcasted tolerance
for $m = 0, 1, ..., N$
    $oldx = x$
    for $i = 0, ..., n-1$
        $S_r = 0$
        for $j = r(n/k^2), ... , (r+1)(n/k^2)-1$
            $S_r = S_r + a_{ij}$
        if ($r = 0$)
            reduce-sum: $S = \sum_{r=0}^{k^2-1} S_r$
            $x_i = (b_i - S - a_{ij})/a_{ii}$
            Send $x_i$ to processor $P_{i(n/k^2)}$
        else
            send partial sum $S_r$ to processor 0
        if ($r(n/k^2) \leq i < (r+1)(n/k^2)$)
            receive $x_i$
    }
    if ($\|x - oldx\| <$ tolerance) exit

Figure 4: The Proposed Parallel Gauss-Seidel Algorithm

Figure 4 outlines the steps of the proposed parallel GS algorithm. Given the problem inputs $A$, $b$, $x$ and tolerance, processor 0 (the master processor) partitions matrix $A$ into blocks of $n/k^2$ columns each and scatters them to the processors. Processor $r$ receives the $r^{th}$ block of the matrix containing the $a_{ij}$ elements for $i$ and $j$ in the ranges: $0 \leq i < n$ and $r(n/k^2) \leq j < (r+1)(n/k^2)-1$, respectively. Then processor 0 scatters the elements of the vector $x$ to the processors. Processor $P_r$ receives the $a_{ij}$ segment of $n/k^2$ elements of the vector $x$, i.e. the $x_j$ elements for $j$ in the range $r(n/k^2) \leq j \leq (r+1)(n/k^2)-1$. After scattering $A$ and $x$, processor 0 broadcasts the value of tolerance to all processors. The rest of the algorithm is similar to the sequential algorithm except that the loop for calculating and summing the $a_{ij}$'s is done in parallel by the $k^2$ processors. Processor number $r$ calculates the partial sum of the $a_{ij}$'s for $j$ in the range $r(n/k^2), ... , (r+1)(n/k^2)-1$ corresponding to the $r^{th}$ block and the $r^{th}$ segment of matrix $A$ and vector $x$, respectively, received by processor number $r$. The partial sums are then collected and summed (reduce-sum) at processor 0 which completes the calculation of the new $x_i$ element. Processor 0 sends the new $x_i$ element to the processor in charge of $x_i$, i.e. processor number $i(n/k^2)$.

4. Analysis of the Parallel GS Algorithm

Table 1 outlines timing expressions for the various steps of the parallel GS algorithm of figure 4. We assume it takes an amount of time $t_{copy}$ to copy the value of a real number from a memory location to another, $t_{multiply}$ to multiply two real numbers, $t_{add}$ to add two real numbers, and $t_{sqrt}$ to calculate the square root of a real number. To send a single data item (a real number) from a processor to another processor on the $k \times k$ torus it takes in the worst case a time $t_{send}$ proportional to the diameter (maximum node-to-node distance) of the $k \times k$ torus which is $2[k/2] = k$. Expressions for $t_{broadcast}$, $t_{scatter}$, and $t_{reduce-sum}$ which correspond to the time required for group communication operations (broadcast, scatter, reduce-sum) on the $k \times k$ torus will be derived later in this section.

Table 1: Execution Time of the GS Algorithm

<table>
<thead>
<tr>
<th>Algorithm Step</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Scatter the blocks of matrix $A$</td>
<td>$T_1 = t_{scatter}(n^2) = O(n^2k)$</td>
</tr>
<tr>
<td>2. Scatter the segments of vector $x$</td>
<td>$T_2 = t_{scatter}(n) = O(nk)$</td>
</tr>
<tr>
<td>3. Broadcast tolerance</td>
<td>$T_3 = t_{broadcast}(1) = O(k)$</td>
</tr>
<tr>
<td>4. Save old $x$</td>
<td>$T_4 = nt_{copy} = O(n)$</td>
</tr>
<tr>
<td>5. Calculate the $S_r$ partial sums</td>
<td>$T_5 = n/k^2(t_{multiply} + t_{add}) = O(nk^2)$</td>
</tr>
<tr>
<td>6. Collect and sum the $S_r$ partial sums</td>
<td>$T_6 = t_{reduce-sum} = O(n)$</td>
</tr>
<tr>
<td>7. Calculate $x_i$</td>
<td>$T_7 = 2(t_{multiply} + t_{add}) = O(1)$</td>
</tr>
<tr>
<td>8. Send $x_i$ to processor number $i(n/k^2)$</td>
<td>$T_8 = t_{send} = O(k)$</td>
</tr>
<tr>
<td>9. Termination Test</td>
<td>$T_9 = n(2t_{add} + t_{multiply}) + t_{sqrt} = O(n)$</td>
</tr>
</tbody>
</table>

It can be seen from figure 4 that the total time required by the parallel GS algorithm is given by:
Now we derive timing expressions for the group communication operations (broadcast, scatter and, reduce-sum) on the \( k \times k \) torus.

The broadcasting of a message of size \( s \) from a source node to all other nodes in the \( k \times k \) torus can be done in time:

\[
t_{\text{broadcast}}(s) = kst_{\text{com}}, \text{ where } t_{\text{com}} \text{ is the time needed to send one real number from one processor to a neighboring processor in the } k \times k \text{ torus. This expression is justified as follows: broadcasting in the } k \times k \text{ torus can be done in two phases. In the first phase the message is propagated horizontally (across the row where the source node is located) in both directions (left and right) starting at the source node and making use of the wrap-around links if needed. This broadcasting phase requires } [k/2] \text{ single-hop communication steps. During the second phase of the broadcasting all the nodes which received the message during the first phase (i.e. all the nodes located on the source row) initiate in parallel vertical propagations of the message across the columns in both directions (up and down) making use of the vertical wrap-around links if needed. This phase also requires } [k/2] \text{ single-hop communication steps. The total time required by this broadcasting algorithm is therefore:}
\]

\[
t_{\text{broadcast}}(s) = 2 \left[ \frac{k}{2} \right] st_{\text{com}} = O(sk) \tag{4}
\]

Similarly to broadcasting, scattering a message of size \( s \) in the \( k \times k \) torus (each processor will receive one chunk of the message of size \( s/k^2 \)) can also be done in two phases except that not the whole message propagates horizontally and vertically. During the first phase when a node on the source row receives a message it extracts one part of size \( sk \) (to be scattered across the column during the second phase) and divides the remaining part in two equal parts and sends one of them to the right neighbor and the other to the left neighbor. The sizes of the horizontally propagated messages are therefore successively: \((s/sk)/2 = s(k-1)/2k, s(k-3)/2k, s(k-5)/2k, \ldots, \) and \((s(k-k-2))/2k = sk, \) assuming without loss of generality that \( k \) is odd. The time for the first phase of the scatter operation is therefore:

\[
\left( \frac{2}{2k} \right) t_{\text{com}}((k - 1) + (k - 3) + \cdots + 2) \approx \frac{s k t_{\text{com}}}{8} \tag{5}
\]

During the second phase of the scatter operation, the same steps can be followed across each column (parallel vertical scattering) starting at the processors of the source row each with an initial message of size \( s' = sk \) to be scattered vertically. Using the same analysis of the first phase yields the timing expression \( s' k t_{\text{com}}/8 = st_{\text{com}}/8 \) for the second phase. The total time for the scatter operation is therefore:

\[
t_{\text{scatter}}(s) = s(k+1)t_{\text{com}}/8 = O(sk) \tag{6}
\]

The reduce-sum group communication operation is the operation of gathering while summing a set of numbers initially scattered at all processors (one number per processor). The final sum is collected at one sink processor. This operation is needed in the parallel GS algorithm to collect and sum up the \( S_i \) partial sums. It can be done by reversing the two phases of the broadcasting operation. During the first phase, partial sums across all columns are calculated in parallel by a sequence of summing and sending up or down (whichever is closer) to the sink row. This yields a set of partial sums stored at the processors of the sink row. During the second phase, the processors on the sink row calculate the final sum by a sequence of summing and sending right or left (whichever is closer) to the sink processor. The final sum will be stored at the sink processor. Each of the two phases requires \([k/2] \) steps of summing and sending. The resulting total time is therefore:

\[
t_{\text{reduce-sum}} = 2 \left[ \frac{k}{2} \right] (t_{\text{com}} + t_{\text{add}}) = O(k) \tag{7}
\]

Going back to expression (3) of the total execution time of the parallel GS algorithm and plugging in the expressions of the broadcasting, scattering and reduce-sum operations we obtain the following expression assuming \( n \) very large compared to \( k \):

\[
T_{\text{total}} = n^2(k+1)t_{\text{com}}/8 + n(k+1)t_{\text{com}}/8 + k t_{\text{com}} + N[n t_{\text{copy}} + n(n/k^2)(t_{\text{multiply}} + t_{\text{add}}) + k(t_{\text{com}} + t_{\text{add}}) + 2(t_{\text{multiply}} + t_{\text{add}}) + t_{\text{copy}}] = O(kn^2 + Nn^2k^2 + Nnk) \tag{8}
\]

If the number of iterations \( N \) and the size of the linear system \( n \) are large compared to the number of processors \( k^2 \) (in the order of \( k^2 \) or larger) then the total running time \( T_{\text{total}} \) of the parallel GS algorithm is \( O(Nn^2k^2) \) which is \( k^2 \) times smaller than the time of the sequential algorithm. We can therefore conclude that when the problem size \( n \) and the number of iterations \( N \) are large compared to \( k \), then the proposed parallel GS algorithm on the \( k \times k \) torus has a near optimal speedup (i.e. speedup nearly equal to the number of processors \( k^2 \)) and hence a near optimal speedup and efficiency as illustrated in figures 5 and 6.

![Figure 5: Speedup of the Parallel GS Algorithm](image)

611
5. Conclusion

This paper has proposed a parallel Gauss-Seidel iterative algorithm for solving large systems of linear equations on a two-dimensional torus network-on-chip architecture. The proposed algorithm makes use of the horizontal and vertical interconnects and wrap-around links of the torus for efficient group communication operations between the processors including broadcasting, scattering and reduce-sum operations. These efficient group communication operations are at the heart of the proposed parallel algorithm. We have shown that the proposed parallel algorithm has near optimal speedup and efficiency when solving large linear systems that require a large number of iterations. The work can be extended to a 3D torus NoC.

References


