SQU Journal for Science, 2015, 20(1), 29-38

An Efficient Parallel Gauss-Seidel Algorithm on a 3D Torus Network-on-Chip

Khaled Day[1]* and Mohammad H. Al-Towaiq[2]

*[1]*Department of Computer Science, College of Science, Sultan Qaboos University, P.O. Box: 36, PC 123, Al-Khod, Muscat, Sultanate of Oman. [2]Jordan University of Sciences and Technology, Jordan.*Email: kday@squ.edu.om.*

ABSTRACT: Network-on-chip (NoC) multi-core architectures with a large number of processing elements are becoming a reality with the recent developments in technology. In these modern systems the processing elements are interconnected with regular NoC topologies such as meshes and tori. In this paper we propose a parallel Gauss-Seidel (GS) iterative algorithm for solving large systems of linear equations on a 3-dimensional torus NoC architecture. The proposed parallel algorithm is $O(Nn^2/k^3)$ time complexity for solving a system with a matrix of order n on a $k\times k\times k$ 3D torus NoC architecture with N iterations assuming n and N are large compared to k. We show that under these conditions the proposed parallel GS algorithm has near optimal speedup.

Keywords: Network-on-chip; 3D torus; Parallel algorithm; Linear system of equations; Gauss-Seidel method.

خوارزمية غاوس سيدال متوازية فعالة باستخدام شبكة تورس على رقاقة ثلاثية الأبعاد

خالد داي و محمد طويق

ملخص: أصبحت الأنظمة المتعددة النواة ذات الشبكة على رقاقة واحدة والتي تحتوي على عدد كبير من وحدات المعالجة حقيقة واقعية بفضل التطورات الحديثة في مجال التكنولوجيا. في هذه الأنظمة المتطورة يتم ربط وحدات المعالجة بشبكات ذات أشكال هندسية منتظمة كشكل التشابك وشكل تورس. نقترح في هذه الورقة خوارزمية غاوس سيدال تكرارية متوازية لحل أنظمة معادلات خطية ذات أحجام كبيرة في زمن معالجة لا يتجاوز حدود $\mathbf{O(Nn^2/k^3)}$ وذلك لحل نظام معادلات خطية يكون فيها عدد المعادلات $\boldsymbol{n}$ باستخدام شبكة تورس ثلاثية الأبعاد بحجم $\boldsymbol{k\times k\times k}$ وعدد تكرار في الخوارزمية $\boldsymbol{N}$ مع فرضية أن قيم $\boldsymbol{n}$ و $\boldsymbol{N}$ كبيرة مقارنة بقيمة $\boldsymbol{k}$. نبرهن أنه في هذه الحالة تكون الخوارزمية المتوازية المقترحة ذات تسريع يكاد يكون الأمثل.

كلمات مفتاحية: شبكة على رقاقة واحدة، شكل تورس ثلاثي الأبعاد، خوارزمية متوازية، نظام معادلات خطية و طريقة غاوس سيدال.

1. Introduction

In this paper we propose a parallel Gauss-Seidel algorithm based on message passing for solving a system of linear equations: $Ax = b$ where A is an n by n dense matrix, b is a known n-vector and x is an n-vector to be determined. Systems of linear equations are of immense importance in mathematics, and to its applications to areas in the physical sciences, economics, engineering, social sciences and biological sciences, among many others. Even complicated situations are frequently approximated by a linear model as a first step. The solution of a system of nonlinear equations is achieved by an iterative procedure involving the solution of a series of linear equations. Similarly, the solution of ordinary differential equations, partial differential equations and integral equations using the finite difference method leads to a system of linear or nonlinear equations. Linear equations also arise frequently in numerical analysis.

There are two classes of methods for solving linear systems of equations: direct and iterative methods. A direct method is a fixed number of operations carried out once, at the end of which the solution is produced. Gauss elimination and related strategies on a linear system is an example of such methods. Direct methods are often too expensive in terms of computation time, memory requirements, or both. As an alternative, linear systems are usually solved with 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].

For early parallel computers such as the CM-2 and the Intel iPSC/860 [2], it was observed that the single iteration steps of most iterative methods offered too little opportunity for parallelism in comparison with, for instance, direct methods for dense matrices. In particular, the inner products required per iteration for many iterative methods were

identified as obstacles because of communication. This has led to attempts to combine iteration steps, or to combine the message passing for different inner products.

The Gauss-Seidel is one of the most efficient iterative methods for solving linear systems that arise in solving partial differential equations. Many parallel implementations have been proposed including those reported in [3-8]. Some implementations have been developed for regular problems such as the Laplace equation [9, 10], circuit simulation problems [11], power load-flow problems [12], and for many applications of inter-dependent constraints, or as a relaxation step in multi-grid methods [3]. In [13] several parallelization strategies for the dense Gauss-Seidel method are presented. These strategies are compared and evaluated through performance measurements on a large range of hardware architectures. The authors found that these new architectures do not offer the same trade-off in terms of computation power versus communication and synchronization overheads as do traditional high-performance platforms. In 1999, Wang and Xu [14] presented a specific technique for solving convection-dominated problems. Their algorithm uses crosswind thin blocks in a block Gauss-Seidel method. Their method is based on a special partitioning technique for a block iterative method for solving the linear system derived from a monotone discretization scheme for convection diffusion problems. They conclude that crosswind grouping is essential for the rapid convergence of the method. In 2005, Grabel *et al.* [15] presented two simple techniques for improving the performance of the parallel Gauss-Seidel method for the 3D Poisson equation by optimizing cache usage as well as reducing the number of communication steps.

In 2006, Nobuhiko *et al.* [16] presented a novel parallel algorithm for the block Gauss-Seidel method. The algorithm is devised by focusing on Reitzinger's coarsening scheme for those linear systems derived from the finite element discretization with first order tetrahedral elements.

The time consumed on communication between processors limits the parallel computation speed. With advances in technology, chips with a large number of cores (processing elements) are becoming a reality. Communication between processing elements in such multi-core 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 [17].

Several topologies have been proposed and studied for NoCs including mesh-based and tree-based topologies [17]. The emerging three-dimensional (3D) integration and process technologies allow the design of multi-level Integrated Circuits (ICs). This creates new design opportunities in NoC design. For example, a considerable reduction can be achieved in the number and length of global interconnections using three-dimensional integration.

Motivated by these new developments in technology and by the resulting improved performance of inter-processor communication on modern 3D network-on-chip systems, we propose, and analyze the complexity of, a new parallel implementation of the Gauss-Seidel algorithm on 3D torus NoC architectures. The proposed algorithm uses message passing for inter-processor communication. It is an extension of our previous similar algorithm on 2D torus NoC [18]. A shorter version of this paper has been presented in [19].

The rest of the paper is structured as follows: in section 2 an introduction is presented including a description of the sequential Gauss-Seidel algorithm to be parallelized, as well as an introduction to the 3D torus NoC architecture. We describe the proposed parallel algorithm in section 3, and we evaluate its performance in section 4. The paper is concluded in section 5.

2. Preliminaries

a. The Gauss-Seidel sequential agorithm

The Gauss-Seidel (GS) algorithm is an improvement of the Jacobi algorithm. GS corrects the i^{th} component $x_i^{(m)}$ of the vector $\boldsymbol{x}^{(m)}$ in the order i = 0, 1, …., n-1. The approximation solution is updated immediately after the new component is determined. The newly computed component $x_i^{(m+1)}$ can be changed within a working vector which is redefined at each relaxation step, and this results in the following iterative formula [10]:

$$x_i^{(m+1)} = (b_i - \sum_{j=0}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{n-1} a_{ij} x_j^{(m)}) / a_{ii} \quad (1)$$

In matrix notation, equation (2) becomes:

$$(D - L)x^{(m+1)} = Ux^{(m)} + b, m \geq 0. \quad (2)$$

In (2) L, D, and U are the lower, diagonal, and upper triangular parts of matrix A respectively. Figure 1 outlines the sequential GS algorithm.

```
Sequential GS Algorithm
{       Input A, b, x, tolerance, N
        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 – sum) / a_ii
                }
                if (||x^(m+1) – x^(m)|| < tolerance)
                {       output the solution x^(m+1)
                        exit
                }
                x^(m) = x^(m+1)
        }
}
```

Figure 1. The sequential Gauss-Seidel algorithm.

It is well known that the GS algorithm will always converge if the matrix *A* is strictly or irreducibly 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*), hence the need for faster parallel implementations.

b. The 3D torus network-on-chip architecture

An *n*-dimensional torus network, also called a wrap-around mesh or toroidal network, is a Cartesian product of *n* cycle networks. Two-dimensional mesh-based topologies (such as 2D mesh and 2D torus) have been the most popular among the known NoC topologies. Their popularity is due to their modularity (they are easily expandable by adding new nodes and links without modifying the existing structure), their ability to be partitioned into smaller meshes, their simple XY routing strategy, and their facilitated implementation. They also have a regular structure and short inter-switch wires. They have been used in several chip multiprocessors such as the RAW processor [20], the TRIPS processor [21], the Intel 80-core Terascale processor [22], the 100-core TILE-Gx100 processor from Tilera [23] and the Single-Chip Cloud Computer (SCC) of Intel [24].

The emerging three-dimensional (3D) integration and process technologies allow the design of multi-level Integrated Circuits (ICs) [25]. This creates new opportunities in NoC design [26, 27]. For example, a considerable reduction can be achieved in the number and length of global interconnections using three-dimensional integration. 3D NoCs are more advantageous than 2D NoCs in providing better performance for large multi-core systems [28]. Long horizontal wires in 2D NoCs can be replaced by very short vertical links in 3D NoCs.

Despite being available for quite a while, the 3D torus architecture now has the potential to be one of the most attractive interconnection topologies for future large NoC systems. This is because nowadays severe challenges are faced due to the rising number of cores (nodes). Petascale and exascale installations require, and will require, hundreds or thousands of cores to efficiently work together. The 3D torus topology offers the ability to add nodes without affecting performance and reliability. It is also important for future large multi-core systems to consume less energy. Connecting nodes using a 3D torus topology means that each node is connected to the adjacent ones via short cabling (except for the wrap-around links) in 6 different "directions": X+, X-, Y+, Y-, Z+, Z-. The pair-wise connectivity between nearest neighbor nodes of a 3D torus helps to reduce energy consumption and communication latency. A 3-dimesional torus interconnection topology is illustrated in Figure 2.

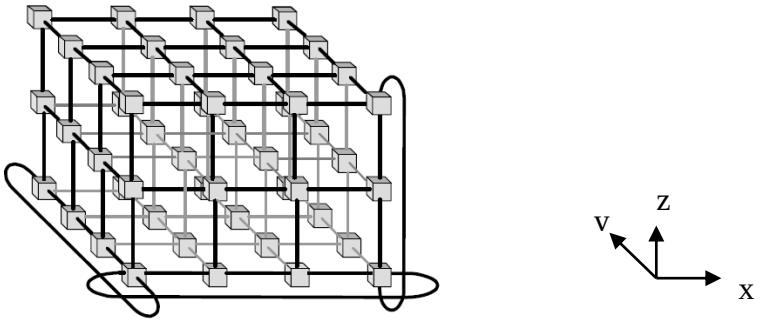

Figure 2. The 3D torus NoC topology.

3. The proposed parallel GS algorithm

In our proposed parallel GS algorithm, we partition the $n \times n$ matrix A into blocks of n/k^3 columns each, and scatter them to the k^3 processors of a $k \times k \times k$ 3D torus. The k^3 processors are identified by (x, y, z) coordinates, $0 \le x, y, z \le k$-1, as illustrated in Figure 2. Processor (x, y, z) is connected to the six neighboring processors $(x$-1, $y, z)$, $(x$+1, $y, z)$, $(x, y$-1, $z)$, $(x, y$+1, $z)$, $(x, y, z$-1) and $(x, y, z$+1). The +1 and -1 operations in these expressions are modulo k in order to include the wrap-around links.

We also assign sequential processor numbers (ids) to the k^3 processors as follows: the processor whose coordinates are (x, y, z) is assigned the sequential id: $\text{id}(x, y, z) = x + ky + k^2z$. In this way, the k^3 processors are also identified with sequential ids in the range 0 ... k^3-1 as illustrated in Figure 3.

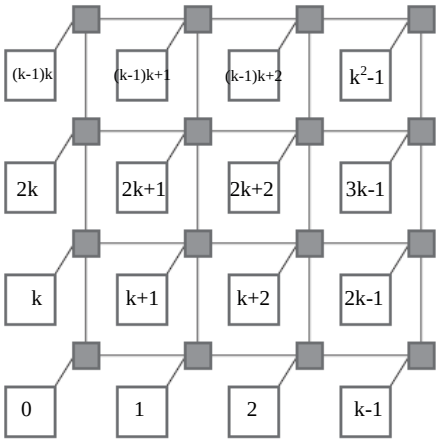

Figure 3. Sequential ids of the processors in plane z = 0.

Figure 4 outlines the steps of the proposed parallel GS algorithm. Given the problem inputs $\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$ and *tolerance*, processor 0 (the master processor) partitions matrix $\boldsymbol{A}$ into blocks of n/k^3 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: $0 \le i < n$ and $r(n/k^3) \le j \le (r+1)(n/k^3)$-1, respectively. Then processor 0 scatters the elements of the vector $\boldsymbol{x}$ to the processors. Processor P_r receives the r^{th} segment of n/k^3 elements of the vector $\boldsymbol{x}$, i.e. the x_j elements for j in the range $r(n/k^3) \le j \le (r+1)(n/k^3)$-1. After scattering $\boldsymbol{A}$ and $\boldsymbol{x}$, processor 0 broadcasts the value of *tolerance* to all processors. The rest of the algorithm is similar to the sequential algorithm, in except that the loop for calculating and summing the $a_{ij}x_j$'s is done in parallel by the k^3 processors. Processor number r calculates the partial sum of the $a_{ij}x_j$'s, $j = r(n/k^3), \ldots, (r+1)(n/k^3)$-1, corresponding to the r^{th} block of matrix $\boldsymbol{A}$ and the r^{th} segment of vector $\boldsymbol{x}$ received by this processor. 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 then sends the new x_i to the processor in charge of x_i, that is $P_{i/(n/k^3)}$.

```
Parallel_GS Algorithm
{   //Let r be the sequential id of local processor (0 ≤ r < k^3)
    if (r = 0) //master processor
    {   Input A, b, x, tolerance
        Partition A into k^3 blocks of n/k^3 columns each
        Scatter the blocks of columns of A to the k^3 processors
        Partition x into k^3 segments of n/k^3 elements each
        Scatter the segments of elements of x to the processors
    } else
    {   Receive the r^th block of n/k^3 columns of A
        Receive the r^th segment of n/k^3 elements of x
    }
    for m = 0, 1, ..., N
    {   if (r = 0) oldx = x
        for i= 0,..., n-1
        {   S_r = 0
            for j = r(n/k^3), ..., (r+1)(n/k^3)-1
                S_r = S_r + a_ij x_j
            if (r = 0)
            {   Gather and sum the partial sums: S = sum_{r=0}^{k^3-1} S_r
                x_i = (b_i - S - a_ii x_i ) / a_ii
                Send x_i to processor P_{i/(n/k^3)}
            }
            else
            {   Send partial sum S_r to processor 0 (contribute to gather)
                if (r = i/(n/k^3) ) receive x_i
            }
        }
        if (r = 0) and (||x-oldx|| < tolerance) terminate computation
    }
}
```

Figure 4. The proposed parallel Gauss-Seidel algorithm.

Notice that this parallel GS algorithm is based on parallelizing the calculation of *sum* inside the inner loop of the sequential algorithm (Figure 1). Each processor is in charge of a distinct block of columns of matrix $\boldsymbol{A}$ and of a distinct segment of vector $\boldsymbol{x}$ elements, allowing it to contribute to summing the $a_{ij}x_j$'s in a distinct range of j. More precisely, processor P_r (i.e. with sequential id r) is in charge of the j range: $r\ (n/k^3) \le j \le (r+1)(n/k^3)-1$. The partial sums calculated in parallel by the different processors are gathered by the master, forcing all processors to synchronize at this point before proceeding to the next i iteration. This yields a correct parallelization of the calculation of *sum* of the sequential algorithm.

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 one 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. Expressions for $t_{broadcast}$, $t_{scatter}$, and $t_{reduce\text{-}sum}$ which correspond to the time required for group communication operations (broadcast, scatter, reduce-sum) in the $k\times k\times k$ torus will be derived later in this section.

Table 1. Complexity of the parallel GS algorithm.

Algorithm Step	Time Complexity
1. Scatter the blocks of columns of ***A***	$T_1 = t_{scatter}(n^2) = O(n^2k)$
2. Scatter the segments of vector ***x***	$T_2 = t_{scatter}(n) = O(nk)$
3. Broadcast *tolerance*	$T_3 = t_{broadcast}(1) = O(k)$
4. Save old ***x***	$T_4 = nt_{copy} = O(n)$
5. Calculate the S_r partial sums	$T_5 = (n/k^3)(t_{multiply} + t_{add}) = O(n/k^3)$
6. Reduce-sum the S_r partial sums	$T_6 = t_{reduce\text{-}sum} = O(k)$
7. Calculate x_i	$T_7 = 2(t_{multiply} + t_{add}) = O(1)$
8. Send x_i to processor $i/(n/k^3)$	$T_8 = t_{send}(1) = O(k)$
9. Termination Test	$T_9 = n(2\ t_{add} + t_{multiply}) + t_{sqrt} = O(n)$

It can be seen from Figure 4 that the total time required by the parallel GS algorithm is given by:

$$T_{total} = T_1+T_2+T_3+N[T_4+n(T_5+T_6+T_7+T_8)+T_9] \quad (3)$$

It remains to derive expressions for $t_{broadcast}$, $t_{scatter}$, and $t_{reduce\text{-}sum}$ which correspond to the timing on the 3D torus of the group communication operations broadcast, scatter, and reduce-sum respectively.

a. The cost of broadcasting on the 3D torus

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

$t_{broadcast}(s) = 3\lfloor k/2 \rfloor s.t_{com}$, where t_{com} is the time needed to send a single number from one processor to a neighboring processor in the $k\times k\times k$ torus. This expression is justified as follows: broadcasting in the $k\times k\times k$ torus can be done in three phases as illustrated in Figure 5. In the first phase, the message is propagated on the X dimension in both directions (X+ and X-), starting at the source node and making use of the wrap-around links on the X dimension if needed. This X broadcasting phase requires $\lfloor k/2 \rfloor$ communication steps.

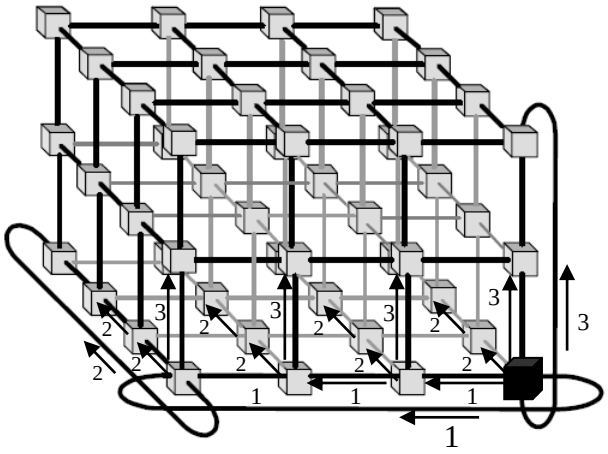

Figure 5. Broadcasting on the 3D torus in 3 phases.

In the second phase of the broadcasting, all the nodes which received the message during the first phase (the nodes located on the source row) initiate parallel propagations of the message across the Y dimension in both directions (Y+ and Y-), making use of the wrap-around links on the Y dimension if needed. This phase also requires $\lfloor k/2 \rfloor$ single-hop communication steps.

In the third phase of the broadcasting, all the nodes which received the message during the first and second phases (i.e. all the nodes located on the source plane) initiate parallel propagations of the message across the Z dimension in both directions (Z+ and Z-), making use of the wrap-around links on the Z dimension if needed. This third phase also requires $\lfloor k/2 \rfloor$ single-hop communication steps. The total time required by this broadcasting algorithm is therefore: $t_{broadcast}(s) = 3\lfloor k/2 \rfloor s t_{com} = \mathrm{O}(sk)$. Figure 6 shows an implementation of this broadcasting algorithm on the $k\times k\times k$ torus.

```
Broadcast(M, source = (x_s, y_s, z_s))
{       //let local = (c_local, y_local, z_local)
        if (local = source)
        {       send M to (x_local + 1, y_local, z_local)
                send M to (x_local - 1, y_local, z_local)
                send M to (x_local, y_local + 1, z_local)
                send M to (x_local, y_local - 1, z_local)
                send M to (x_local, y_local, z_local + 1)
                send M to (x_local, y_local, z_local - 1)
        }
        else
        {       receive M    //let sender = (x_sender, y_sender, z_sender)
            if(y_local = y_source) and (z_local = z_source) // phase 1
                {                       if(x_sender = x_local – 1)
                                send M to (x_local + 1, y_local, z_local)
                send M to (x_local, y_local + 1, z_local)
                send M to (x_local, y_local - 1, z_local)
                send M to (x_local, y_local, z_local + 1)
                send M to (x_local, y_local, z_local - 1)
                }
            else if (z_local = z_source) // phase 2
                if(y_sender = y_local – 1)
                                send M to (x_local, y_local + 1, z_local)
                else        send M to (x_local, y_local - 1, z_local)
                                        send M to (x_local, y_local, z_local + 1)
                                        send M to (x_local, y_local, z_local - 1)
                else // phase 3
                if(z_sender = z_local – 1)
                                        send M to (x_local, y_local, z_local + 1)
                else        send M to (x_local, y_local, z_local - 1)
        }
}
```

Figure 6. Implementation of the 3D torus broadcasting algorithm.

b. The cost of scattering on the 3D torus

Similarly to broadcasting, scattering a message of size M in the $k \times k \times k$ 3D torus (each processor will receive one chunk of the message of size M/k^3) can also be done in three phases, except that not the whole message propagates in the three phases. During the first phase, when a node receives a message, it extracts its part of size M/k (to be scattered across the Y dimension during the second phase) and divides the remaining part into two equal parts and sends one of them to the X+ neighbor and the other to the X- neighbor. The sizes of the messages propagated in this way on the X dimension are therefore successively: $(M\text{-}M/k)/2 = M(k\text{-}1)/2k$, $M(k\text{-}3)/2k$, $M(k\text{-}5)/2k$, …, and $M(k\text{-}(k\text{-}2))/2k = M/k$, assuming without loss of generality that k is odd. The time for the first phase of the scatter operation is therefore: $(M/2k)t_{com}((k-1)+(k-3)+...+2) \cong Mkt_{com}/8$.

During the second phase of the scatter operation, the same steps can be followed across the Y dimension starting at the processors of the source row, each with an initial message of size $M' = M/k$. Using the same analysis as of the first phase yields the timing expression $M'kt_{com}/8 = Mt_{com}/8$ for the second phase. A similar calculation gives the timing expression $Mt_{com}/8k$ for the third phase. The total time for the scatter operation is therefore $t_{scatter}(M) = M(k+1+1/k)t_{com}/8 = O(Mk)$.

c. The cost of reduce-sum on the 3D torus

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_r partial sums. It can be done by reversing the three phases of the broadcasting operation. During the first phase, partial sums across dimension Z are calculated in parallel by a sequence of summing and sending on the Z+ or Z- direction (whichever is closer) to the sink plane. This yields a set of partial sums stored at the processors of the sink plane. During the second phase, the processors on the sink plane calculate partial sums across dimension Y in parallel by a sequence of summing and sending on the Y+ or Y- direction (whichever is closer) to the sink row. This yields a set of partial sums stored at the processors of the sink row. The processors on the sink row then calculate the final sum by a sequence of summing and sending on the X+ or X- direction (whichever is closer) to the sink processor. The final sum will be stored at the sink processor. Each of the three phases requires $\lfloor k/2 \rfloor$ steps of summing and sending. The resulting total time of the Reduce-Sum operation is therefore: $t_{reduce\_sum} = 3\lfloor k/2 \rfloor (t_{com} + t_{add}) = O(k)$.

d. Overall cost of the parallel GS algorithm

Notice that T_1, T_2, T_3, T_6, and T_8 in expression (3) of the total execution time of the parallel GS algorithm correspond to communication steps, while T_4, T_5, T_7 and T_9 correspond to computation steps. Using the obtained timing expressions of the broadcasting, scattering and reduce-sum operations, the total communication time T_{comm} is given by the following:

$$T_{comm} = T_1 + T_2 + T_3 + Nn[T_6 + T_8] = O((kn^2 + Nnk)t_{com})$$

Notice that T_{comm} is proportional to t_{com} (the single hop communication cost) which is much faster on NoC networks than on cluster or multiprocessor networks. Therefore the proposed parallel GS algorithm runs faster on a NoC than on a loosely coupled cluster or a tightly coupled multiprocessor.

The total computation time T_{comp} is given by:

$$T_{comp} = N[T_4 + n(T_5 + T_7) + T_9] = O(Nn^2/k^3)$$

The complexity of the total execution time T_{total} of the parallel GS algorithm is therefore:

$$T_{total} = T_{comm} + T_{comp} = O(kn^2 + Nnk + Nn^2/k^3) \quad (4)$$

When the size of the linear system n and the number of iterations N are large compared to the number of processors k^3 (in the order of k^4 or larger), the total execution time T_{total} of the parallel GS algorithm in expression (4) is dominated by the term Nn^2/k^3 which is k^3 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, the proposed parallel GS algorithm has a near optimal speedup (nearly equal to the number of processors k^3) and hence a near optimal efficiency. Remember that the speedup of a parallel algorithm is defined as the time of the sequential algorithm divided by the time of the parallel algorithm, while the efficiency is defined as the speedup divided by the number of processors. Figures 7 and 8 illustrate how the speedup and efficiency increase as n and N increase.

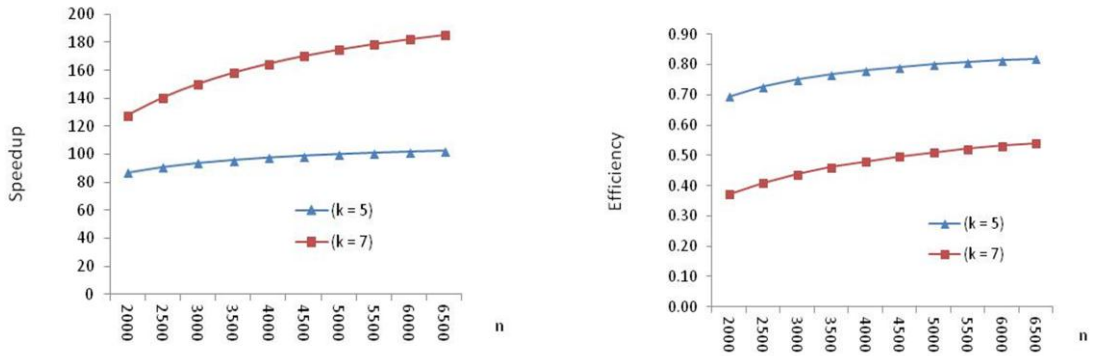

Figure 7. Speedup and efficiency of the parallel GS algorithm as a function of n.

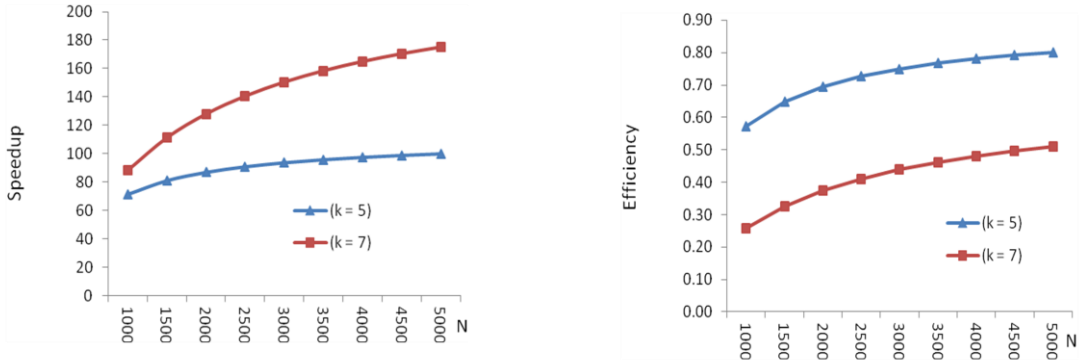

Figure 8. Speedup and efficiency of the parallel GS algorithm as a function of *N*.

5. Conclusion

We have proposed a parallel Gauss-Seidel iterative algorithm for solving large systems of linear equations on a 3D torus network-on-chip architecture. The proposed algorithm makes use of the X, Y, Z interconnects with wrap-around links of the 3D 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 algorithm. We have shown that the proposed parallel algorithm has near optimal speedup when solving large linear systems that require a large number of iterations. This work can be extended by an experimental or simulation-based performance evaluation of the proposed parallel algorithm.

References

1. Adams, L. and Xie, D. New Parallel SOR Method by Domain Partitioning, *SIAM J. Sci. Comp*., 1999, **20(22)**, 2261-2281.
2. Krystynak, J. and Nitzberg, B. Performance Characteristics of the iPSC/860 and CM-2 I/O Systems, *Proceedings of the Seventh International Parallel Processing Symposium*, Newport, CA, 13-16 April 1993, pp. 837-841.
3. Adams, M.F. A Distributed Memory Unstructured Gauss-Seidel Algorithm for Multigrid Smoothers, *Proc. of* 2001 *ACM/IEEE Conference on Supercomputing*.
4. Hu, C., Zang, J., Wang, J., Li, J. and Ding, L. A New Parallel Gauss-Seidel Method by Iterative Space Alternate Tiling, 16th international Conference on *Parallel Architecture and Compilation Techniques*, PACT 2007, 15-19 September.
5. Murugan, M., Sridhar, S. and Arvindam, S. A Parallel Implementation of the Gauss-Seidel Method on the Flosolver, *Technical Report, National Aeronautical Labaratory, Bangalor*, India, July 2006.
6. Olszewski, L. A Timing Comparison of the Conjugate Gradient and Gauss-Siedel Parallel Algorithms in a One-dimensional Flow Equation Using PVM, *Proc. of the 33rd Annual Southeast Regional Conference*, Clemson, South Carolina, 1995, pp. 205-212.
7. Thongkrajay, U. and Kulworawanichpong, T. Convergence Improvement of Gauss-Seidel Power Flow Solution Using Load Transfer Technique, *Proceeding* (296) *Modeling, Identification, and Control-2008*, Feb. 11-13, 2008, Innsbruck, Austria.
8. Wallin, D., Lof, H., Hagersten, E. and Holmgren, S. Multigrid and Gauss-Seidel Smoothers revisited: Parallelization on Chip Multiprocessors, *Proceedings of ICS06 Conference*, June 28-30, 2006, Cairns, Queensland, Australia.
9. Fox, G., Johnson, M., Lyzanga, G., Otto, S., Salmon, J. and Walker, D. *Solving Problems on Concurrent Processors*, Printice Hall, 1988.
10. Golub, G. and J.M. Ortega *Scientific Computing with an Introduction to Parallel Computing*, Academic Press, Boston, MA, 1993.
11. Saleh, R.A., Gallivan, K.A., Chang, M., Hajj, I.N., Smart, D. and Trich, T.N. Parallel Circuit Simulation on Supercomputers, *Proc. of the IEEE*, 11989, **77(12)**, 1915-1930.
12. Wallch, Y. *Calculations and Programs for Power System Networks*, Printice Hall, 1986.
13. Courtecuisse, H. and Allard, J. Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors, *High Performance Computation Conference (HPCC), IEEE Press*, June 2009.
14. Wang, F. and Xu, J. A crosswind Block Iterative Method for Convection-Dominated problems, *SIAM J. Sci. Comp*., 1999, **21(2)**, 620-645.
15. Grabel, J., Land, B. and Uebertholz, P. Performance Optimization for the Parallel Gauss-Seidel Smoother, *PAMM*, 2005, **5(1)**, 831-832.

16. Nobuhiko, O., Takeshi, M. Takeshi, I. and Masaaki, S. A Parallel Block Gauss-Seidel Smoother for Algebraic Multigrid Method in Edge-Element Analysis, *Papers of Technical meeting on Static Apparatus, IEE Japan*, 2006, **Vol. 6**, no. 58-61, 63-75, pp. 55-60.
17. Benini, L. and Micheli, G.D. *Networks on Chips: Technology and Tools*, Morgan Kaufmann, 2006.
18. Al-Towaiq, M. and Day, K. Parallel Gauss-Seidel on a Torus Network-On-Chip Architecture, *J. Interconnection Networks*, 2012, **13(1, 2)**, 1-14.
19. Day, K. and Al-Towaiq, M. A Parallel Gauss-Seidel Algorithm on a 3D Torus Network-on-Chip Architecture, 10th HiPEAC Conference on High Performance and Embedded Architecture and Compilers, January 19-21, 2015, Amsterdam, Netherlands.
20. Taylor, M.B., Lee, W., Amarasinghe, S. and Agarwal, A. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures, *Int'l Symposium on High-Performance Computer Architecture (HPCA)*, 2003, pp. 341-353, Anaheim, California.
21. Gratz, P., Kim, C., Sankaralingam, K., Hanson, H., Shivakumar, P. and Burger, S.K. On-Chip Interconnection Networks of the Trips Chip, *IEEE Micro*, 2007, **27(5)**, 41-50.
22. Hoskote, Y., Vangal, S.R., Singh, A., Borkar, N. and Borkar, S. A 5-GHZ Mesh Interconnect for a Teraflops Processor, *IEEE Micro*, 2007, **27(5**), 51-61.
23. Ramey, C. TILE-Gx100 ManyCoreProcessor: Acceleration Interfaces and Architecture, *Hot Chips 23*, Stanford, CA, August 17, 2011.
24. Howard , J. *et. al*, A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling, *IEEE Journal of Solid-State Circuits*, January 2011, **46(1)**, 173-183.
25. Feero, B.S. and Pande, P.P. Networks-on-Chip in a Three-Dimensional Environment: A Performance Evaluation, *IEEE Trans. on Computers*, January 2009, **58(1)**, 32-45.
26. Pavlidis, V.F and Friedman, E.G. 3-D Topologies for Networks-on-Chip, *IEEE TVLSI*, 2007, **15(10)**, 1081-1090.
27. Marcon, C. *et. al*., Tiny NoC: A 3D Mesh Topology with Router Channel Optimization for Area and Latency Minimization, *Proc. 27th International Conference on VLSI Design*, 5-9 Jan 2014, Mumbai, India, 2014, pp. 228-233.
28. Kourdy, R. and Nouri, M.R. Performance Comparison of 2D and 3D Torus Network-on-Chip Architectures, *Journal of Computing*, 2012, **4(2)**, 119-122.

Received 26 October 2014
Accepted 15 February 2015