Solving Linear Algebraic Equations by using Distributed Memory Programming

Oğuzhan Çiftçi
Dept. of Electrical & Electronics Engineering
Marmara University
Istanbul, Türkiye
0000-0002-9763-3677

***Abstract*—In this paper, we study solving of linear algebraic equations with the form of $Ax = b$ in a distributed manner by using distributed memory programming. We consider open-source message passing interface (MPI) approach to model our simulations. Also we review the sequential programming approach and make comparison with MPI on the performances under different network topologies.**

***Index Terms*—Multi-core Systems, Algebraic Equations**

I. Introduction

Parallel programming techniques are widely used in computer simulations to enhance the computational performance. Nowadays, the computing systems are mostly designed as having multiple cores in their architectures and these cores can be used for parallel programming. One of the main benefits of having the multiple cores in computing systems is that these cores can be utilized individually at the same time. Therefore, programmers have the opportunity to use these cores a networked system. In the networked systems, each core processes their workloads and exchange their results with their connected neighbors[**b1**]–[**b5**].

There are numerous types of techniques to achieve parallel programming structure. One of the widely used techniques consider that each core processes their workload individually and has its own local memory in the computing system. Therefore, each core can be modelled as an individual processor and data in the local memory of the cores should be explicitly exchanged between the other cores. This structure is known as distributed memory systems. The Message Passing Interface (MPI) is a powerful tool to utilize the distributed memory programming.

On the other hand, most of the programmers do not spend enough time to benefit from the parallel programming tools [**b6**]. In order to draw attention into parallel programming techniques, we suggest a use case for integrating these tools with the conventional multi-core computing systems.

Linear algebraic equations (LAEs) are very essential in order to analyze systems mathematically in many different fields such as Engineering, Economics, and Architecture. We usually use the following form to show the algebraic equations:

$$Ax = b \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$ is the system matrix, $x \in \mathbb{R}^n$ is the solution vector, and $b \in \mathbb{R}^m$ is the observation vector[**b7**].

There are numerous different algorithms suggested to solve such systems, however, they can be investigated in two major topics: Centralized algorithms and Distributed algorithms. Centralized algorithms generally consider the system altogether and try to manipulate the system matrix in order to make it invertible. On the other hand, distributed algorithms use a different approach. The main idea for the distributed algorithms is partitioning the system matrix into many parts and solve these parts individually by using different solvers. Then collect the solutions from each solver in accordance with a connection network between the solvers. Finally, solvers have a consensus on a solution for the overall system $Ax = b$ if there is a solution[**b7**].

In this paper, we mainly focus on solving linear algebraic equations of the form (1) by using the distributed memory programming technique in C language. We use the MPI model which was designed to utilize the distributed memory architecture in multi-core computing systems.

This paper is organized as follows. In the following section, we provide mathematical preliminaries to form a network and introduce an algorithm to solve linear algebraic equation systems in a distributed manner. In the next section, we form numerical algebraic equation systems to be simulated in a multi-core computer. We introduced the pseudocodes for solving such equation systems both in sequential programming and distributed memory programming techniques. Section 5 compares the sequential programming and distributed memory programming simulation results. In section 6, we conclude and discuss the results provided in this paper.

II. Mathematical Preliminary

In a multi-core computing system, we consider each core as an individual processor as mentioned previously. However, the data exchange relationship should be well-defined to have distributed memory structure at hand. Communication channels between the agents in a multi-core network can be expressed mathematically by using graph theory. We consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as illustrated in Fig. 1, where $\mathcal{V} = \{1, 2, \ldots, m\}$ is the set of nodes and $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the set of directed edges between the nodes. We assume node i and node j are adjacent to each other if a communication link exists between them, i.e., $(i, j) \in \mathcal{E}$. These nodes are also called *neighbors*. Furthermore, we define the set of nodes that

sends information to node i as in-neighbors of node i, i.e., $\mathcal{N}_i^+ = \{j \in \mathcal{V} : (j,i) \in \mathcal{E}\}$.

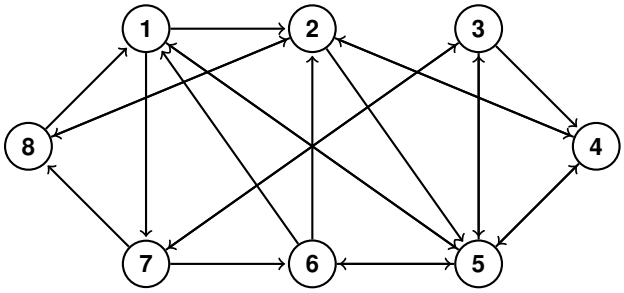

Fig. 1. A directed graph $\mathcal{G}$.

The nature of the distributed algorithms for solving algebraic equations fits into the parallel programming structure. Especially, when we consider each core in the CPU as a node of a network, we can directly benefit from the parallel programming techniques in order to simulate the solution of the algebraic equations by using an appropriate algorithm. There are many different algorithms that can be utilized to solve the linear algebraic equation of the form of (1) distributively. However, we use a discrete-time distributed algorithm suggested in Mou et al., 2015 such that

$$\boldsymbol{x}_i(t+1) = \boldsymbol{x}_i(t) - P_{ker(A_i)} \sum_{j \in \mathcal{N}_i^+(t)} w_{ij}(t)\Big(\boldsymbol{x}_i(t) - \boldsymbol{x}_j(t)\Big) \tag{2}$$

where $w_{ij}(t)$ is the weighting coefficient which was designed to be $\left|\mathcal{N}_i^+(t)\right|^{-1}$, $\forall\ i\ \in\ \mathcal{V}$, $P_{ker(A_i)}$ is the projection matrix onto the kernel of A_i, and $t \geq 1$.

III. Problem Definition

In this section, we define two different linear algebraic equation systems numerically. The first system has the following equations:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + 3x_2 + x_3 &= 3 \\ x_1 - x_2 - 2x_3 &= -6. \end{aligned} \tag{3}$$

Above algebraic equation system has a unique solution of $\mathbf{x} = [-1, 1, 2]^T$. In order to solve a linear algebraic equation by using (2), we form a network with three nodes as illustrated in Fig. 2 and it satisfies all the conditions defined in Mou et al., 2015.

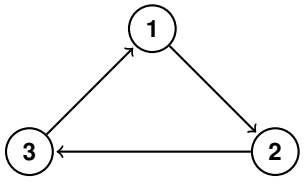

Fig. 2. A directed graph with three nodes.

As described in Mou et al., 2015, each node in the network should know a subset of the system (3). In addition, each node must begin its iteration in algorithm (2) with a solution to its equations.

Secondly, we design a system with eight equations of the form of (1) as follows:

$$\begin{bmatrix} 4 & -2 & 3 & 2 & 1 & 6 & 3 & 0 \\ 6 & 0 & 1 & -1 & -3 & 0 & -3 & -1 \\ -1 & -3 & 6 & -2 & 5 & -1 & -3 & 4 \\ 1 & 2 & -2 & -1 & -2 & -3 & 4 & -3 \\ 1 & 0 & 4 & 5 & -3 & -1 & 6 & -3 \\ 4 & -2 & 4 & -3 & 0 & -3 & 2 & 3 \\ 5 & -1 & 2 & -1 & 1 & 2 & -2 & 3 \\ -2 & 6 & -2 & -3 & -2 & 4 & 5 & 2 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 33 \\ 23 \\ 11 \\ 2 \\ 13 \\ 12 \\ 23 \\ 7 \end{bmatrix} \tag{4}$$

We suggest the network illustrated in Fig. 1 as the multi-core system to solve above equation system. Although it has infinitely many solutions, we can still use algorithm (2) to achieve consensus among the cores as a solution to overall system (4).

Furthermore, we simulate different randomly generated equation systems for simulation time comparison. Each system is guaranteed to have at least one solution and we utilize the network topologies for the multi-core systems illustrated in Fig. 1 and Fig. 2 in accordance with their equation numbers.

IV. Distributed Memory Programming

In this section, we briefly introduce the distributed memory programming concept. However, we first consider the sequential programming approach. In sequential programming, we use a single core for simulations. In our particular problem of solving linear algebraic equations, the sequential programming can be easily applied. We develop a sequential C program for solving linear algebraic equations distributively.

In this code snippet, we first define the problem of interest in the form of (1) in line 19. Then we defined the connection relationship between the nodes to utilize data produced by the neighboring node. Next, we directly use the algorithm defined in (2) for each node. In here, we used *rank* as the current processed node. All these processes are done by a single core and each node waits for others to finish their computations.

Now, we study the distributed memory programming. Essentially, a distributed memory system contains multiple core and local memory pairs connected by a network. Each core in this system processes its own work in accordance with its clock. However, when we need to send and receive data from one node to another to accomplish a complex task, we need to utilize a message passing protocol between the corresponding cores. One of the most used standard on distributed memory systems is MPI. We simulate the problems we defined in the previous section by using MPI on a distributed memory system.

We develop the program in Listing 2 by utilizing the MPI standard in C language to solve linear algebraic equations in a distributed manner. In the given program, we first initialize the MPI with the number of cores we have in the distributed

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define ERROR ... // tolerance
#define N ... // number of agents

// Prototypes
void  problemDefinition();
int   determineInNeighbor(int rank);
void  algorithmMou(int rank, float x_neighbor[N]);
float norm(float soln[], float unique[], int
↪  num_rows);
void  printVector(float vector[N], int size);

int main(int argc, char** argv) {
int rank;
float soln_norm;
float x_neighbor[N];
problemDefinition();

do{
        for (rank = 0; rank < N ; rank++){
        x_neighbor =
        ↪  determineInNeighbor(rank);
        algorithmMou(rank, x_neighbor);
        soln_norm = norm(x_next[rank], x[rank], N);
        }
}while(soln_norm > ERROR);
printVector(x_next[0], N*N);
return EXIT_SUCCESS;
}
```

Listing 1: A sequential program for solving LAEs.

TABLE I
COMPUTATION TIMES.

	LAE in (3)	**LAE in** (4)
Sequential Program	418 μsec	36245 μsec
Distributed-memory Program (MPI)	34169 μsec	40223 μsec

system in lines between 21 and 23. Then, *MPI_Bcast* in line 24 is used to broadcast initial data to all cores to be used in the system. Next, each core exchanges data between its neighboring cores defined in the network. It is important to note that, each core processes its received data simultaneously until the tolerance is achieved. Then *MPI_Gather* collects data from each core into the designated destination core in line 33.

V. RESULTS

In this section, we review the results for the LAEs defined Section III by using sequential and distributed-memory programs defined in Listing 1 and 2.

In Table I, we provided the computation times for solving LAEs in (3) and (4) under the network topologies illustrated in Fig. 1 and Fig. 2. The values given in the table are the average computation times of 150 simulations in the same computer.

As can be observed from the table, the computation time needed to solve the problem (3) with sequential program is explicitly less than we have with distributed-memory program. On the other hand, we achieved consensus for problem (4) almost at the same average time for both approaches.

These results mainly show that the distributed-memory system structure superior the sequential approach when the

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define ERROR ... // tolerance
#define N ... // number of agents

// Prototypes
void  problemDefinition();
int   determineInNeighbor(int rank);
int   determineOutNeighbor(int rank);
void  algorithmMou(int rank, float x_neighbor[N]);
float norm(float soln[], float unique[], int
↪  num_rows);
void  printVector(float vector[N], int size);

int   main(int argc, char** argv) {
int   rank, size, tagno = 101, iter;
float soln_norm, result[N*N], x_neighbor[N];
problemDefinition();
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Bcast(x_next, N * N, MPI_FLOAT, 0,
↪  MPI_COMM_WORLD);
do{
        neighbor_in = determineInNeighbor(rank);
        neighbor_out =
        ↪  determineOutNeighbor(rank);
        MPI_Send(&x_next[rank], N, MPI_FLOAT,
        ↪  neighbor_out, tagno,
        ↪  MPI_COMM_WORLD);
        MPI_Recv(&x_neighbor, N, MPI_FLOAT,
        ↪  neighbor_in, tagno, MPI_COMM_WORLD,
        ↪  MPI_STATUS_IGNORE);
        algorithmMou(rank,
        ↪  x_neighbor);
        soln_norm = norm(x_next[rank],
        ↪  x_current[rank], N);
}while(soln_norm < ERROR);
MPI_Gather(&x_next[rank], N, MPI_FLOAT, &result[rank],
↪  N, MPI_FLOAT, 0, MPI_COMM_WORLD);
if (rank == 0) {
    printVector(result, N*N);
}
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Listing 2: A distributed-memory program for solving LAEs.

underlying topology gets complicated. The main reason for that is a node in distributed-memory system do not necessarily has to wait to process its data until the process of all the other nodes. On the contrary, each node must wait for the others in sequential approach since it uses a single core.

VI. CONCLUSION

In this paper, we explored distributed-memory programming with MPI for solving linear algebraic equations in a distributed manner. We also reviewed the sequential approach for the same problem of interest. We compared the simulation times with two different problems under different network topologies for both approaches. Although the network topology is getting complicated when we have a larger system, simulations on the distributed-memory systems may be preferable. For future work, it would be convenient to operate with more comprehensive multi-core computing systems such as super computers in TRUBA in order to simulate more complicated systems. Also, we would like to study other parallel programming techniques to assess the performance more accurately.

REFERENCES

[1] Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L. & Chapman, B. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*. **37**, 562-575 (2011)

[2] LeBlanc, T., Subhlok, J. & Gabriel, E. A high-level interpreted MPI library for parallel computing in volunteer environments. *2010 10th IEEE/ACM International Conference On Cluster, Cloud And Grid Computing*. pp. 673-678 (2010)

[3] Imamura, T., Tsujita, Y., Koide, H. & Takemiya, H. An architecture of Stampi: MPI library on a cluster of parallel computers. *Recent Advances In Parallel Virtual Machine And Message Passing Interface: 7th European PVM/MPI Users' Group Meeting Balatonfüred, Hungary, September 10–13, 2000 Proceedings 7*. pp. 200-207 (2000)

[4] Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T. & August, D. Revisiting the sequential programming model for the multicore era. *IEEE Micro*. **28**, 12-20 (2008)

[5] Eijkhout, V. Teaching distributed memory programming from mental models. *Journal Of Parallel And Distributed Computing*. **118** pp. 120-127 (2018)

[6] Hikosaka, O., Nakahara, H., Rand, M., Sakai, K., Lu, X., Nakamura, K., Miyachi, S. & Doya, K. Parallel neural networks for learning sequential procedures. *Trends In Neurosciences*. **22**, 464-471 (1999)

[7] Mou, S., Liu, J. & Morse, A. A distributed algorithm for solving a linear algebraic equation. *IEEE Transactions On Automatic Control*. **60**, 2863-2878 (2015)