Chapter -2

Literature Survey

In this chapter, we review the published research work related to our work. The work reviewed involves the studies on parallel computing, and existing parallel algorithm of solution of system of linear equations, numerical integration, and polynomial interpolation. Experimental observations of various algorithms which are related to my problems are reviewed and there is discussion of some of the problems.

2.1 Introduction

In the latter half of the 20th century, encouraged by the increase in computing power, science and industry began asking questions with unparalleled complexity that ultimately known as grand challenges [27]. These problems cover a range of topics, examples include solution of large systems of simultaneous equations, numerical integrations, polynomial interpolations, and simulations of complex systems (e.g. weather prediction). These problems share a common feature in that their applications would bring advances in theory and science. Another common feature for these problems is that they are tractable but inefficiently without using a great deal of computing power. To solve these problems efficiently, I turned towards design of parallel algorithms and their performance evaluations using shared memory on Multi-core system.

Unfortunately, it became obvious that the rate of growth of computing power would be insufficient to support continuous work on the challenges. Hence, both sciences and industry interested in acquiring the needed power in next to no time than the general predictions. The simplest way to express these predictions is by using Moore’s law, which (roughly) states that computing power approximately doubles every 18 months [20]. Moore correlated computing power with “Cramming more components into integrated circuits” (Moore). By considering throughout history, computing power of both conventional and multi-core computers has been based on the performance of a single processor, increasing it usually involved fitting more components onto the chip. Moore’s statement implies that if we cannot cram more components onto a single processor,
without introducing a fundamentally new approach, computing power will decline. It was not long before research groups such as Cray Research, IBM, and Control Data Corporation (CDC), concluded that if it becomes harder to fit more components onto a single processor, the next step could simply be cramming more processors together, thus the idea of parallel computing began to develop. While this idea sounds simple, the actual transition from conventional to parallel computing is hardly simple and trivial [39, 54].

Conventional computing and programming have traditionally been sequential or serial, meaning that one instruction was executed at a time, using a single processor. In Flynn’s taxonomy, this is known as SISD (single instruction, single data) architecture. The classical example is the Von Neumann machine, consisting of a single CPU and memory with a bus between them. From early on, serial computing on a SISD machine became more popular than parallel computing, because the input and output (I/O) mechanisms are straightforward compared to the more complex I/O operations needed for a parallel machine. In addition, serial programming tends to be more intuitive and thus “easier” than parallel programming. Even without considering the aforesaid difficulties in speeding up a single processor, serial computing still suffers from the bottleneck of the classical Von Neumann machine. The bottleneck is usually defined as the limitation of the transfer rate between the CPU and memory. “No matter how fast is our CPU, the speed of our programs is limited by the transfer rate of set of instructions and data between memory and CPU [20].

Parallel programming provides a strategy for solving problems in a non-sequential manner by splitting the data and processing it into multiple concurrent “threads.” In Flynn’s classification this is known as MIMD computing (multiple instruction, multiple data). Even though the benefits of parallel processing were recognized early on, building a parallel machine was not as easy as constructing a sequential machine. There were many approaches for creating a system that supports multiple instructions and multiple data. While all the approaches offered a solution to the bottleneck problem and bypassed Moore’s law, they still suffered from the same drawbacks as their predecessors - vastly different operating systems and custom programming languages [39].
However, in the 1990s, several factors combined to change the way parallel computing was done. The first factor was the availability of Open Source operating systems that could be tailored and distributed freely like Linux. Next followed open APIs, for parallel programming that standardized much of the implementation process and made parallel programs portable. Finally, the decreasing cost of networking components, as well as off-the-shelf CPUs, contributed to the success in massively parallelizing conventional processors. Sterling and Becker were among the first to come up with the idea of using multiple conventional computers to build a parallel computer. Thus, the Beowulf project was born [20].

Thanks to the growing popularity of Open Source software, the new generation of Beowulf parallel machines had a common OS and a common parallel API. Essentially, a Beowulf parallel machine, also known as a "cluster", consists of many "off the shelf" computers, known as nodes, connected in a small LAN that communicate data and instructions amongst each other by passing messages across the network. At first, it was thought that the traffic caused by message passing between each node would cause network congestion. However, the message passing APIs are continually optimized, compressing and simplifying the messages but OpenMP solve this problem. Thus, quite a few of the advantages of serial programming, such as the low cost, the portability, and the simpler development process, finally made their way into the parallel world.

In general, parallel programming can be achieved in two ways, implicitly and explicitly. The implicit methods only require the writer to specify a higher-level implementation of the program and then the parallel code will be produced by the compiler. For example, an implicitly parallel program could be implemented in High Performance Fortran. The explicit model depends on the programmer implementing the parallelism in the program by means of directly specifying the threads and controls [20].

The advantage is that the explicit parallel model can be applied to a wider range of problems and it tends to produce programs that are more portable. There are a few tools available for explicit parallel programming. The one of the most common model for parallel programming is shared memory like OpenMP. Other models include message passing specifically MPI, the Message Passing Interface library (Pacheco; Mpi-Forum)
Parallel Virtual Machines (PVMs). Unlike the MPI, OpenMP use the shared memory with multiple CPU.

Memory plays an important part in computing; hence it is worthwhile to discuss, briefly, memory models in parallel computing. Some systems are based on shared memory, others on distributed memory. In the systems that utilize shared memory, multiple processors access the same memory medium, while in distributed memory systems, each processor has its own memory. One advantage of distributed memory systems is that memory is scalable with the number of processors. If we adhere to the distributed memory model, when the number of processors increases, the memory can increase in proportion. Memory access times are rapid, as every processor accesses its memory without the interference of the other processors.

In addition, distributed memory systems are easier to assemble, since commodity systems (off-the-shelf) are easily adaptable to create a distributed system. Their low cost and relatively high performance give them significant advantages. Because these systems are linked together by a network, communication is accomplished by sending messages between each node. This can become a disadvantage if there is a large amount of data passed between nodes across the network, given that the limit of the network speed significantly decreases the memory bandwidth and increases latency. Shared memory systems on the other hand tend to be more complex. In this model, all of the available memory appears as a single physical address space to all the processors. This greatly increases memory bandwidth and decreases latency. There are other two costs that need to be considered in shared memory systems: consistency and coherency between memory and cache. Locks and semaphores need to be used to protect areas of memory, so that multiple processors cannot modify the same memory address [7, 20, 22].

Our study concerns with the design of parallel algorithm for some numerical problems which has been described in chapter 1. Hence we restrict the detailed review to research in domain of numerical problems. This chapter is organized as follows: In section 2 we discussed the prior research and research papers on parallel algorithms of numerical problems. Section 3 focuses on studies on parallel algorithm for numerical problems and section 4 provides the comparative studies in the domain of research.
2.2 Prior Researches on Parallel Numerical Algorithms

All numerical problems are mainly concerned with how to write the parallel algorithm that compute the solution efficiently. Design of parallel algorithm for these problems is also one of the major concerns and hence substantial researches have also been done. Here we have a chronology overview of the work carried out in parallel algorithm for numerical problem domain.

*Bertsekas* and *Tsitsiklis* have possibly been the first one to design the parallel algorithm for solving system of linear equations $Ax=b$, where $x$ is unknown vector to be determine [8]. They proposed the direct method to find the exact solution with finite number of operations, typically for $O(n^3)$. The authors theoretically discussed the concept of complexity and efficiency. They presented the parallel algorithm for Gauss elimination method and found that the algorithm can be executed on linear array of $n$- processors in time $O(n^2)$.

*McGinn* and *Shaw* have proposed the parallel algorithm for Gauss elimination in both the shared memory and distributed memory environment [37]. For the implementation they used the technology *OpenMP* and *MPI*. They presented the parallel LU and Gaussian algorithm for linear system of equations and analyze the results of various load balancing schemes on two platforms. They found that the major disadvantage in distributed environment for this application is that most of distribution of work is data related, not task related. The only way that the node can access the data is by passing it back and forth. In shared environment, this is not a problem, because the data is made available to all the processors at all the time. Due to this parallel algorithm using shared environment is better than the distributed environment.

*Ghodsi, Bahman* and *Rahni* have proposed a Parallel Implementation of Gram-Schmidt Algorithm for Dense Linear System of Equations [24]. They presented a parallel algorithm based on QR- factorization called direct method for solution of system of linear equations. They presented the orthogonalization and triangularization procedures to verify the efficiency of algorithm. They implemented their algorithms using high performance library *MPI*. The QR- factorization computing time is $O(n^3)$. They also proposed that
whatever the number of equation increases, the performance of equation become well and more processor can be used.

**Rajalakshmi** has proposed the Parallel Algorithm for Solving Large System simultaneous Linear Equations [42]. She presented the parallel and sequential algorithm for solving the equations using Gauss Elimination and Gauss Jordon method. In this paper author used the java language, because it has support of parallel execution using multithreading. Author has also presented the execution time of sequential and parallel algorithms. The details of this paper have been discussed in section 2.3.

**Igumenov** and **Petkus** have proposed an Analysis of Parallel Calculations in Computer Network [32]. They found the complex problems, whose solutions can’t obtain efficiently using only one computer. The solution of those complex problems can be solved efficiently by several computers connected to network. They presented the classical parallel solution algorithms and analysis of clusters that consist of 60 processors. They calculated an approximate value of Pi, matrix and vector multiplications, solution of linear equations using the algorithms and analyses its performances. They found that parallel algorithms are distributed into several classes: data parallel algorithm, functional parallel algorithm and master-slave type algorithm. The master-slaves apply to all tasks, where list of task are not interdependent. The task distribution among computers is major problem in construction of parallel algorithm. In this task distribution among computers is major problem. Author presented algorithms that are based on master-slave. The computer master generates and save the list of tasks and distributes to computer slaves, after calculating received task, the computer slaves send the results back to master, which store the common result and supplement the list of tasks.

Authors used the software **MPICH v.1.2.5** for implementation and solved parallel problems in computer network and analyzed the parallel algorithms in terms of speedup. They used **100 MB** network and **1 GB** network and found that, in order to increase the efficiency of algorithms it requires faster network. The author concluded that realization of parallel algorithms accelerate solution of problems compared with serial algorithm.
Schurer has proposed an Adaptive Numerical Integration on Message Passing System [46]. He described the parallel adaptive numerical algorithms for approximation of a multidimensional integration over hyper rectangular regions. They proposed two methods: parallel algorithm using global region collection and using local region collection. The adaptive algorithm is based on embedded cubature rule used to approximate the integral. The algorithm divides the initial region into sub regions and repeats this process on sub regions with the largest estimated error until termination criteria. They found that the maintaining the collection of sub regions and keeping it stored is easy in sequential program. However, if multiple processing nodes are working on approximation of integral, it is difficult to maintain the region collection. Using one global region, collection result is bottleneck, because all other nodes have to access the data structure by communicating with the manager node. By using local region, collections avoid global communications. Thus the local region collection algorithm results in better since the global communication is avoided in it. As a result they found that the algorithm with global region collection performs poorly with two processors, because only one node is evaluating the integral, while other node is maintaining region collection and performance is identical to sequential algorithm. With more processors, global algorithm performs better than the local algorithm.

Bull and Freeman have proposed A Comparison of Parallel Algorithms for Multidimensional Integration [9]. In this the author presented the single multi-list algorithm to approximate the multidimensional integration to a specified absolute accuracy. They used two approaches to parallel numerical integration. One is based on adopting the idea of sequential globally adaptive algorithms to parallel context by selecting a number of sub-regions of the range of integration. The other approach proceeds by imposing initial static partitioning of the range (region) of integration and treats the resulting sub-problems as independent, and therefore capable of concurrent solution. The key difference between these two approaches is that the first maintain the single list of sub-region and second maintain independent list for each processors. The multi-list approach is better than single list approach because it can be implemented in either a message passing or a shared memory programming paradigm, whereas single list approach is suitable for only shared memory. Multi-list makes better use of data locality and thus has lower communication overhead than single list approach.
Ahmadi, Naeini and Sarbazi-azad have been proposed the Efficient SIMD Numerical Interpolation Technique [1]. They presented the SIMD model and how the various interpolation methods can be used to gain the speedup, how these methods implicitly allow parallelism that affect the speedup. They used the interpolation methods using the vector processing capability embedded in Pentium processors. They implemented four interpolation methods using SSE (streaming SIMD extension) technology and then analyze their speedup.

Sarbazi-Azad, Mackenzie, Ould-Khaoua, and Min have proposed An Efficient Parallel Algorithm Lagrange Interpolation and its Performance [43]. Author presented a parallel algorithm for computing N point interpolation on Cube Connected Cycles (CCCn) and implemented the algorithm using distributed memory model relies on all-to-all broadcast communication at some stages during computation. The algorithm compute \( y = f(x) \) on a cube connected cycles, given the set of points \((x_0, y_0), (x_1, y_1), \ldots, (x_{N-1}, y_{N-1})\) and value \( x \), where \( N = n2^n \). The computation is carried out in three phases: first the set of points to be interpolated are allocated to the nodes, one point for each node. Then the Lagrange polynomial \( L_i(x); (0 \leq i \leq N-1) \) are computed. Finally the sum of terms \( y_i \cdot L_i(x); (0 \leq i \leq N-1) \) is calculated to obtain the final result, \( y = f(x) \). The algorithm computes on N point’s interpolation requiring communication and computation operations and shows near to optimal speedup.

Zelina has proposed Parallel Algorithm for Computing the Lagrange Interpolation on an Extended Fibonacci Cube [56]. Author found that when number of points \( N \) is very large, a long computation time and a large storage capacity is required to carrying out the calculation. Author mentioned that Goertzel [26] has already introduced a parallel algorithm for a tree topology with \( N \) processors. The parallel algorithm is based on algorithm described in [44] for computing \( N = !n \), node Lagrange interpolation on \( n \)-star graph. The parallel algorithm presented carries out the computation of \( N \)- point Lagrange interpolation in an Extended Fibonacci Cube on total time of \( O(N) \).
2.3 Details of Related Proceedings Studies on Numerical Algorithms

To the research on the parallel algorithm for numerical problems, presented above, we focus on the work of McGinn and Shaw [37], Ghodsi, Mehri and Rahni [24], and Rajalaskshmi [42]. They all proposed methods for solving the system of linear equations. Igumenov and Petkus [32], Schurer [46], Bull and Freeman [9] presented the parallel algorithm for numerical integration using different approaches. Ahmadi, Naeini and Sarbazi-azad [1], Sarbazi-Azad, Mackenzie, and Ould-Khaoua, Min [43], and Zelina [56] presented parallel algorithm for Lagrange Interpolation. Since these studies deals with parallel algorithms for numerical problem which lies in the same domain, we described the detailed study about some of the problems, which are given below:

In the first study, McGinn and Shaw [37] presented a parallel algorithm for Gaussian elimination in both a shared memory environment using, OpenMP, and in a distributed memory environment, using MPI. They studied Parallel LU and Gaussian algorithms for linear system of equations and presented the results of examining various load balancing schemes on both platforms.

Given a system of linear equations as $Ax = b$, there are several different methods to obtain a solution. If a unique solution is known to exist, and the coefficient matrix is full, a direct method such as Gaussian elimination is usually selected. There are several papers that emphasize various parallel approaches for solving a system of linear equations with Gaussian elimination [2, 3, 16, 38].

In this paper, author was concerned with examining the effect of different load balancing schemes available with OpenMP in a shared memory environment and on a distributed platform where MPI was used as the message passing interface. Some work has been done on load balancing for Gaussian elimination such as the article by Howe and Bratcher [31] which compares cyclic and block mapping schemes. A good parallel algorithm for Gaussian elimination is difficult, because of the inherent dependencies in the algorithm, plus the corresponding load balancing issues. They execute both versions of the algorithm on an IBM RS/6000 SP. This machine has 4 distributed nodes, where each node consists of 4 processors contained within a shared memory environment [35].
With this machine, user has the ability to run programs exclusively within the shared environment, or within the distributed environment, or user can run programs that take advantage of both. They designed the parallel program uses OpenMP to distribute the work among the processors in a shared memory environment which is given below. Various load balancing schedules affect the performance of the resulting code and are specified at runtime with a schedule clause.

```c
!$omp parallel do private (xmult) schedule (runtime)
do i = (pivot+1), n
    xmult = a (i, pivot) / a (pivot, pivot)
do j = (pivot+1), n
    a (i, j) = a (i, j) - (xmult * a (pivot, j))
end do
b (i) = b (i) - (xmult * b (pivot))
end do
!$omp end parallel do
end do
```

With a static scheme and a specified chunk size, each processor is statically allocated chunk iterations. The allocation of iterations is done at the beginning of the loop, and each thread will only execute those iterations assigned to it. Using static without a specified chunk size implies the system default chunk size of $n/p$. In dynamic scheme, each thread is allocated a chunk of iterations at the beginning of the loop, but the exact set of iterations that are allocated to each thread is not known.

The guided scheme allocates a system dependent chunk of iterations between threads at the beginning of the loop. It is similar to dynamic scheduling such that once a thread has completed its work it is allocated a new chunk of iterations. The difference is that the new chunk size of iterations decreases exponentially as the iterations available decreases to a specified minimum chunk size [5]. If no chunk size is specified, the minimum is 1.
In his results author has compared the time spent on the forward elimination phase of the problem. The backward substitution phase runs extremely fast, and is a very minimal factor in the time it takes to run this program [31]. Thread communication time has not been extracted from the algorithm. When using a parallel do directive in OpenMP, a do loop must immediately follow the directive. Because of this restriction, a timer would have to be inserted within the do loop and would affect performance. The authors ported their algorithm to a distributed environment and used MPI to allocate the work across multiple processors. One of the major aspects of implementing the Gaussian elimination algorithm on a distributed memory system is the communication time. This has an effect on the resulting performance of the algorithm.

The major disadvantage in using the distributed memory architecture for this application, is that most of the distribution work is data related, not task related. The only way the nodes can access the data is by passing it back and forth. In a shared memory model, it is not a problem, because the data can be made available to all processors at all times. The author has pointed out that the impact on performance occurs as we change the size of \( n \). When we increase the value of \( n \), OpenMP based program displays an improvement in performance as opposed to the MPI program. It is possible that as \( n \) increases, we may find a point where the shared environment will show a greater increase in performance than the distributed platform.

In the second study, Ghodsi, Mehri and Taeibi-Rahni [24] presented a parallel implementation of gram-schmidt algorithm for dense linear system of equations. They presented a parallel algorithm based on QR- factorization called direct method r solution of system of linear equations. Their main aim was to design the parallel algorithm of orthogonalization and triangularization method to find the solution and to verify the efficiency of algorithm. In the field of linear algebra, there are some high performance libraries which are used in scientific purpose. The most famous libraries are: BLAS, LAPACK and SCALAPACK. The BLAS and LAPACK consist of linear algorithm and SCALAPACK is package of parallel algorithm.

They consider a system of linear equations as \( AX=B \), in which \( A \) is a dense and non-singular matrix with non-zero elements. A QR factorization of \( A \in \mathbb{R}_{m \times n} \) is a
factorization $A=QR$, where $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix and $R \in \mathbb{R}^{n \times n}$ is an upper triangular and finally they used back substitution method to find the solution. If $A=QR$ is a QR Factorization of matrix $A \in \mathbb{R}^{n \times n}$, then this linear system can be solved as follow:

$$q_i = \alpha_i + \frac{r(i,i)}{|\alpha_i \cdot q_1|} q_1 + \ldots + \frac{r(i-1,i)}{|\alpha_i \cdot q_{i-1}|} q_{i-1},$$

(1)

So, $Q = \{q_1, q_2, q_3, \ldots, q_n\}$.

The upper triangular matrix $R$ is computed from equation (1) simultaneously. The matrix $R$ is shown as,

$$R = \begin{bmatrix}
1 & r_{1,2} & r_{1,3} & \ldots & r_{1,n} \\
0 & 1 & r_{2,3} & \ldots & r_{2,n} \\
0 & 0 & 1 & \ddots & \ddots \\
\vdots & \vdots & \ddots & \ddots & 1
\end{bmatrix}.$$  

(2)

Multiply the transpose of $Q$ to the both side of system of linear equations,

$$Q^T Q R X = Q^T B.$$  

(3)

Consider $Q^T Q = Q'$, which is a diagonal matrix and then $Q' R = R'$. In the other side, $Q^T B = B'$. So,

$$R' X = B'.$$  

(4)

In this equation, $R$ is an upper triangular matrix. Therefore the unknown matrix $X$ is computed easily by back substitution.

**Parallel Algorithm**

They presented how to solve a system of linear equations with numerical modified Gram-Schmidt QR-factorization method; initially the orthogonal and upper-triangular matrices must be computed. According to the Equation (1), the columns of the orthogonal matrix are computed one-by-one. In the computation of each column, the previous
computed columns of orthogonal matrix are used. When a column of matrix $Q$ is computed, automatically a column of the upper-triangular matrix is known as,

$$
\begin{align*}
  r(i, j) &= 1 & \text{if } i = j \\
  r(i, j) &= 0 & \text{if } i > j \\
  r(i, j) &\text{ From Eq. 1}
\end{align*}
$$

(5)

It seems that, the solution is completely arranged in sequence. It is considered that the denominator of each coefficient, in the computation of a column of orthogonal matrix in equation (1), consists of a term, which has been repeated in the computation of previous column, i.e. $|q_k \cdot q_k|$. On the other hand, when a column of orthogonal matrix $q_i$ is obtained, those terms in the next unknown columns (i.e. from $q_{i+1}$ to $q_n$), which are related to current column can be computed simultaneously. This is a parallel characteristic.

Author has presented that in order to parallelize this algorithm, first of all, the orthogonal matrix is set equal to the matrix $A$ in all processors, because the first term in equation (1) is $a_i$. Then, each processor has responsibility for computing some columns of orthogonal matrix. When a processor completely computes a column, then it broadcasts the column for all other processors and sends its computed coefficients $r(1: i-1, i)$ to the main processor. Therefore, all required data for computing the next terms of remaining columns is available for all processors. This processing continue until, all columns of orthogonal matrix are computed. Afterward, the main processor computes the unknown matrix $X$ sequentially, with back substitution procedure from existing orthogonal and triangular matrix. They implemented the algorithm on a parallel system structure and distributed memory system [15]. The library of parallel processing used in this paper is standard MPI [51] and programming language FORTRAN. The machine used was Pentium Dual-Core 3.00 GHz, 2 MB cache, 2 GB RAM etc.

The important thing in parallelizing an algorithm is efficiency of the original algorithm. The algorithms discussed by the authors perform mathematical tasks that transform an initial data into a desired result using an ordered list of arithmetic operations, comparisons, and decisions. On computers, computation time was heavily
dominated by evaluating floating point operations. If $n$ measures the input matrix $A \in \mathbb{R}^{n \times n}$, then an $O(n^p)$ algorithm is one that, for some positive constant $c$, performs $cn^p$ plus a sum of lower powers of $n$ floating point operations [11]. The computational work involved in a QR-Factorization is of order $2n^3 + O(n^2)$. The efficiency of Gram-Schmidt QR-factorization is $O(n^3)$, which is acceptable in comparison with other method.

Finally, in this paper, author demonstrated a parallel algorithm based on the Gram-Schmidt QR-factorization, which can be used for direct solution of an arbitrary system of linear equations. His purpose was not to compare the efficiency of this method with other solution of system of linear equations; it showed just the limitation and ability of described parallel method.

In third important study, Rajalakshmi has presented parallel algorithm for solving large system of simultaneous equations [42]. Solving system of linear equations is probably one of the most visible applications of linear algebra. In his paper she presented the parallel and sequential algorithms for solving the equations $AX = B$, where $A$ is coefficient matrix, $B$ is a known vector and $X$ are unknown vector.

As a language, author used the java language. Since java is a popular language and has support for parallel execution using the feature of multithreading. She presented the two methods: Gauss elimination and Gauss Jordon for solving large system of linear equations. She also presented the sequential and parallel algorithm and implemented using the multithreaded mechanism of java language. Finally, she presented the results of execution time of sequential and parallel algorithm of both the methods, which is given below in the Table 2.1 and Table 2.2.

**Table 2.1** Compare the execution time in Gauss elimination method.

<table>
<thead>
<tr>
<th>Size of Equations</th>
<th>Execution Time (milliseconds)</th>
<th>Execution time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>45</td>
<td>7</td>
</tr>
</tbody>
</table>
Table 2.2 Compare the execution time in Gauss Jordon method.

<table>
<thead>
<tr>
<th>Size of Equations</th>
<th>Execution Time (milliseconds)</th>
<th>Execution Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>47</td>
<td>8</td>
</tr>
</tbody>
</table>

In this paper, author’s aim was to solve the system of linear equations using sequential and parallel algorithm and comparison of the speedup. The above table shows that the parallel algorithm has good speed up and less time complexity than the sequential algorithm. Similarly, Gauss elimination method is better than Gauss Jordon.

In the fourth study, Rudolf Schurer has presented adaptive numerical integration on message passing systems. He studied algorithms with centralized global region collection and using local region collection. They consider the problem of estimating the multidimensional integral

\[
\text{If := } \int_{C^n} f(x) \, dx,
\]

for a given function \( f: \mathbb{C}^s \rightarrow \mathbb{R} \), where \( \mathbb{C}^s \) denotes a \( s \)-dimensional hyper-rectangular region \([a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_s, b_s]\) subset of \( \mathbb{R} \).

The Adaptive algorithms based on an embedded cubature rule are used to approximate the integral. These algorithms divide the initial region into sub regions and repeat this process on the sub region with the largest estimated error until a certain termination criterion is met. Maintaining the collection of sub regions and keeping it sorted is easy in a sequential program. However, if multiple processing nodes are working on the approximation of the integral, maintaining this region collection becomes a problem.

Author found that using one global region collection \([15, 21]\) results in bottlenecks, because all other nodes have to access this data structure by communicating with the manager node. Using local region collections \([10, 25, 36]\), global communication can be avoided. The algorithms presented by author are given below:
**Sequential Algorithm**

The key concept of this algorithm was to apply a basic cubature rule successively to smaller sub-regions of the original integration domain. The selection of these sub-regions adapts to difficult areas in the integration domain by refining sub-regions with large estimated errors:

**step 1.** The basic rule is applied to the whole integration domain to estimate the value of the integral and the error of this approximation. An embedded degree 7 rule due to Genz and Malik [23] has been used in programs. It includes a degree 5 rule for error estimation.

**step 2.** The region and the estimations for result and error are stored in the region collection. A heap, ordered by the estimated error, is used to implement this collection efficiently.

**step 3.** The region with the largest estimated error is taken from the region collection.

**step 4.** This region is split into two sub-regions by halving it along the coordinate axis where the integrand has the largest local absolute fourth difference [23]. Calculating this fourth divided difference is embedded in the basic rule, so no extra integrand evaluations have to be performed.

**step 5.** Estimations for result and error for these sub-regions are calculated.

**step 6.** Both regions are stored in the region collection.

**step 7.** Goto step 3.

The loop terminated either when a certain absolute or relative error is reached, when the number of integrand evaluations exceeds an upper bound, or if some other criterion is met.

**Parallel Algorithm using a global region collection**

The algorithm is straight-forward parallelization of the sequential algorithm. There is a dedicated manager node maintaining the (global) region collection. When one of the other nodes becomes idle, the manager sends it a new region to be processed. The worker node calculates estimations for its value and error and returns these results to the manager, who stores them in the region collection. If further processing is required, the
manager sends a new region to the worker. Otherwise, a message signalizing termination is sent.

Starting this algorithm with only one sub region is inefficient, because nodes would be idle until the number of sub regions reaches the number of worker nodes. One common approach to avoid this is to each node executes the sequential algorithm until the number of sub regions reaches the number of working nodes. The approach taken is different: Each worker node performs the following steps to find its initial region:

step 1. Set level to 0. Set region to the initial region
step 2. Split region in two parts by halving it along the coordinate axis with the largest fourth divided difference of the integrand.
step 3. If bit number level in the binary representation of the node number is 0, choose the left sub region. If it is 1, choose the right sub region.
step 4. If there are other nodes working on the same sub region (this can be determined easily using bit operations involving the node number and the number of nodes), increment level and continue with step 2.
step 5. Start integration algorithm with region.

The number of iterations of this loop is approximately $\log_2 n$, with $n$ denoting the number of worker nodes. The manager node has to calculate the initial regions for all worker nodes. This task requires $n-1$ calculations of a fourth difference. For a large number of nodes, this can be substantially more expensive than the work required on the worker nodes. However, the worker nodes have to apply the basic rule on their region before they contact the manager for the first time.

**Parallel Algorithm using local region collections**

The main disadvantage of the parallel algorithm with a global region collection is that it does not scale to a large number of processing nodes. Due to the fact that there is only a single manager node which has to serve all worker nodes, this node will become the bottleneck if the number of nodes increases.
To improve scalability, all global communication has to be removed. This implies that there can’t be a dedicated manager node. To balance workload, some communication between processing nodes is required. However, the communication of a certain node can be restricted to a small number of nodes in its neighborhood.

The author’s basic idea of this algorithm was that every node executes the sequential algorithm on a subset of sub regions of the whole integration domain. At the beginning, the algorithm outlined in the previous section is used by each node to find its initial sub region. When it is found, each node uses the sequential algorithm and its own (local) region collection to split regions into sub regions and to adapt to difficulties found there.

They implemented all the algorithms in C++ using MPI for inter-process communication. They found that the algorithm with global region collection perform poorly with two processors, because only node is actually evaluating the integral, while other node is maintaining the region collection. The performance with two processors is almost identical to the performance of sequential algorithm. With the 16 processors, the global algorithm performs better than the local algorithm; however the global algorithm does not scale and break down even for small number of nodes, where the basic rule evaluated quickly and region collection become large.

In next important study, Ahmadi, Naeini, and Sarbazi-Azad have presented efficient SIMD numerical interpolation techniques [1]. In this study the authors have presented how the various interpolation methods can gain speedup using SIMD (Single Instruction Multiple Data) computational model. The fact that how interpolation methods, implicitly allow parallelism that affect the speedup. They discussed the four methods: Lagrange, Newton Gregory Forward, Gauss Forward and B-Spline. For each of these methods, they designed the generic vector algorithm and then they implemented in assembly and C++ for Pentium processor.

The SIMD processing model allows performing a given operation on a set of data instead of single data. This computational model is now supported by many new processors. The authors have focused their implementations on one of the most popular
technology, Intel **SSE** technology. The packed single precision floating-point instructions in **SSE** technology perform **SIMD** operations on packed single precision floating point operands. Each source operand contains four single precision floating-point (32) bits value and destination operand contains results parallel operation performed on corresponding values of two source operands.

In this paper authors discussed the four methods and found that the Lagrange method is the appropriate method to be executed on vector processors. The simplicity of sequential implementations makes this method to be used in many interpolation applications. Lagrange Interpolation, for a set of given points \((x_i, y_i), 0 \leq i \leq 1\), in point \(x\) is carried out as

\[ f(x) = \sum_{i=0}^{n-1} l_i(x) \]  

(1)

Where, \(l_i(x)\) is called Lagrange polynomial, and is given by:

\[ l_i(x) = \prod_{0 \leq j \leq n-1, j \neq i} \frac{(x-x_j)}{(x_i-x_j)} \]  

(2)

It requires a long computation time to carry out the above computation in sequential form. Hence, this interpolation method is usually implemented on a number of parallel systems with different topology, such as the **k-array n-cube** [45].

Authors have presented the vector processing algorithms and implemented using Pentium 4 **SSE** instructions. For Lagrange interpolation, first the common dividend in all Lagrange factors \(l_i(x)\) is computed. Next for each factor the divisor is calculated. After computing Lagrange’s factors, the final value is calculated by equations (1). All operations involved in these steps efficiently exploit **SIMD** operations in **SSE**.

Newton-Gregory forward and Gauss forward method is based on difference table. But we are more concerned about the Lagrange interpolation. For the Newton Gregory forward method they begin with computation of difference table. Then each \(\Delta f_i\) value is multiplied by \((\frac{x}{i})\) and added to S and sum of calculated term is computed. They adapt the
same technique for Gauss Forward method as well. These two methods used a sequential summation procedure to carry out \( f(x) \) and therefore parallelism is exposed only in computing the difference table because of great data dependence between steps in these methods. Thus these two methods are not efficient.

The authors have implemented their algorithms on Intel’s Pentium 4 processor using single precisions floating point. They computed the speedup with two different implementations: the SIMD C++ and assembly code and their equivalent sequential SIMD code. They analyzed the performance of implemented methods as function of number of points being implemented. To compute the speedup for a given number of interpolated points, execution time of SIMD code is divided by execution time of equivalent sequential code. They found that speedup grow as number of points increases. In this paper, authors aim was to design and implement SIMD code for four interpolation methods namely *Lagrange, Newton-Gregory forward, Gauss forward* and *B-Spline*, using Intel’s SSE extension in Pentium 4 processors. They found that the Lagrange method can achieved high speed up when executed in SIMD model. In our research work, the parallel algorithms for numerical problems performance are better than the sequential algorithm and it also scale with the number of processor.