High Performance Numerical Linear Algorithms

Ph.D. Research Seminar

by

Gaurav Trivedi
99407602

under the guidance of

Prof. H. Narayanan
and
Prof. Madhav P. Desai

Department of Electrical Engineering
Indian Institute of Technology, Bombay

May 2000

Abstract

High performance computing is needed for evaluating computationally "bulky" problems. For years, computers have progressed in architecture and hardware aspects to exploit parallelism, but the algorithmic or software aspects are yet to be discovered fully in many fields. There are many aspects related to high performance computing, which we need to know before developing algorithms. Linear algebra lies at the heart of most calculations in scientific computing. Thus there is a need for developing computationally "rich" algorithms for linear algebra. In this report we present some ways to exploit parallelism for achieving high performance for linear algorithms, along with a summary of parallel processing and architecture overview with performance evaluation analysis.

Contents

List of Figures

Chapter 1

Introduction

In the past decade, the world has experienced one of the most exciting periods in computer development. Computer performance improvements have been dramatic - a trend that promises to continue for the next several years. One reason for the improved performance is the rapid advancement in microprocessor technology. Microprocessors have become smaller, denser and more powerful. The result is that microprocessor based super computing is rapidly becoming the technology of preference in attacking some of the most important problems of science and engineering. To exploit microprocessor technology, vendors have developed high parallel computers [2,9].

Highly parallel systems offer the enormous computational power needed for solving some of the most challenging computational problems such as circuit simulation incorporating various effects. Unfortunately, software development has not keep pace with hardware advances. New programming paradigms languages, scheduling and partitioning techniques, and algorithms are needed to fully exploit the power of these highly parallel machines.

A major new trend for scientific problem solving is distributed computing. In distributed computing [10], computers are connected by a network are used collectively to solve a single larger problem. Many scientists are discovering that their computational requirements are best served not by a single, monolithic computer but by a variety of distributed computing resources, linked by high speed networks. By parallel computing, we mean a set of processes that are able to work together to solve a computational problem. There are a few things that are worthwhile to point out. First, the use of parallel processing and the techniques that exploit them are now everywhere; from the personal computer to the fastest computer available. Second, parallel processing doesn't necessary imply high performance computing.

1.1 Traditional computers and their limitation

The traditional computer, or conventional approach to computer design involves a single instruction stream. Instructions are processed sequentially and the result is movement of data from memory to functional unit and back to memory. As demands for faster performance increased, modifications were made to improve the design of the computers. It became evident that a number of factors were limiting potential speed: the switching speed of the devices, packaging and interconnection delays, and compromises in the design to account for realistic tolerances of parameters in the timing of individual components. Even if a dramatic improvement could be made in any of these areas, one factor still limits performance: the speed of light. Today's supercomputers have a cycle time on the order of nanoseconds. One nanosecond translates into the time it takes light to move about a foot (in practice, the speed of pulses through the wiring of a computer ranges from 0.3 to 0.9 feet per nanosecond). Faced by this fundamental limitation, computer designers have begun moving in the direction of parallelism.

In this report we explore some of the issues involved in the use of high performance computing and parallel processing aspects. The organization of the report is given below.

1.2 Organization Of the Report

1. Chapter 2 describes the fundamentals of parallel processing and issues related with high performance computing [1,2,9,10].

2. Chapter 3 describes the techniques used to decrease overhead and improve performance[10].

3. Chapter 4 describes the performance analysis for uniprocessor and parallel processors[2,10].

4. Chapter 5 describes parallel algorithms for solving linear equations[10].

5. Chapter 6 discusses future work that is to be done.

Chapter 2

Parallel Processing and Distributed Computing

Most systems are single processor systems: that is they have only one main CPU. However there is a trend towards multiprocessor systems [2,10]. Such systems have more than one processor in close communication, sharing the computer bus, the clock and sometimes memory and peripheral devices. These system are referred to as *Tightly Coupled Systems* [1,2,10]. A recent trend in computer system is to distribute computation [10] among several processors.

In contrast to the tightly coupled model the processors don't share memory or a clock. Instead each processor has its own local memory. The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. These systems are usually referred to as *Loosely Coupled Systems or Distributed Systems*[1,2,10]. The processors in a distributed system may vary in size and functions. They may include small microprocessors[9], workstations, minicomputers and large general purpose computer systems. These processors are referred to by a number of different names, such as, *sites, nodes, computers,* and so on, depending on the context in which they are mentioned.

The term parallel processing [1,2,9,10] is used in a very general sense to cover methods that involve a deliberate attempt to increase speed by performing computations simultaneously or in parallel. Like any type of processing, parallel processing can be viewed at various level of complexity. At the gate level, for example, a distinction is made between serial arithmetic, which involves computing numbers one at a time, and parallel arithmetic, in which all bits of a number are calculated simultaneously. At the register level, the basic unit of the information is the word, so one can distinguish serial machines, which compute one word at a time from parallel machines which can compute several words simultaneously. Finally, at the processor level where the information unit is a block of words e.g. a program of data set, a parallel machine can process several blocks of information simultaneously. Nowadays, the term parallel computer is reserved for two types of machines:

1. Multiprocessors, i.e. computer with more than one CPU
2. Computers with single CPU that is capable of executing several instructions or computing several distinct data items simultaneously.

Here the term parallel processing is referred for a CPU or the portion thereof capable of processing more than one instruction or set of operands simultaneously at the register level. Increasing the level of parallelism in a processor increases its potential operating speed. The amount of hardware required also increases and with it the cost of the system. But due to the recent technologies cost comes down as the system performance increases. The two most common strategies used in designing computers to get the high performance are to increase the number of functional units or pipes in the processor or to increase the number of processors in the system. A third strategy is a hybrid approach of the about two. In the multiple pipe strategy, more than one pipe is available for the arithmetic operations, but now multi-function pipes are available in the processors, where the same hardware can perform more than one type of arithmetic operation, commonly both an addition and multiplication operation. To utilise this a very high degree of sophistication is required in the compiler. Another approach is to use multi-processor strategy, which is described in the following section with proper classification of the parallel processors. Using this strategy, throughput is increased via multiple job streams. Today, two approaches to multiprocessing are being actively investigated: SIMD(Single Instruction Stream Multiple Data Stream) and MIMD (Multiple Instruction Stream Multiple Data Stream). SIMD and MIMD are described in the next section.

2.1 Type of parallel processor

A typical processor(CPU) operates by fetching instructions and operands from main memory, executing the instructions, and placing the results in the main memory. The instructions can be viewed as forming the instruction stream flowing from main memory to the CPU, while the operands form another stream, the data stream flowing between the processor and the memory.

M.J.Flynn [2,9] has made an informal but useful classification of processor parallelism based on the number of simultaneous instruction and data streams seen by the processor during program execution. Suppose the processor P is operating at its maximum capacity so that its full degree of parallelism is being exhibited. Let $m_I(m_D)$ denote the minimum number of distinct instruction (data) streams which are being actively processed in any number of seven basic steps [2,9]. m_I and m_D are termed the instruction and data stream multiplicities of P and measure its degree of parallelism. Note that m_I and m_D are defined by the minimum number of streams at any point, since the most constrained components of the system (its bottlenecks) determines the overall parallel processing capabilities.

Computers can be roughly divided into four major groups based on the values of m_I and m_D for their CPUs.

1. **Single Instruction Stream Single Data Stream (SISD):** $m_I = m_D = 1$. Most conventional machines with one CPU containing a single arithmetic logical unit capable only of scalar arithmetic fall into this category. This is shown in fig 2.1.

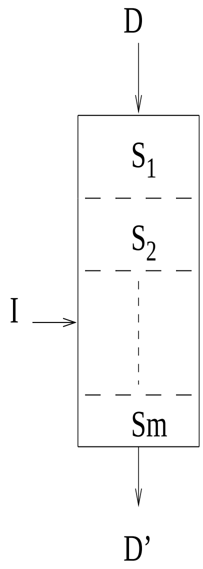

Figure 2.1: SISD Organization

2.**Single Instruction Stream Multiple Data Stream (SIMD):** $m_I = 1$, $m_D > 1$. This category includes machines with a single program control unit and multiple execution units. It also includes associative or content addressable memory processors. In such machines many stored data items may be accessed and processed simultaneously. This is shown in fig 2.2. SIMD machines [2,9,10]permit explicit expression of parallelism in a program. Program segments that cannot be converted into parallel executable form are sent to the processing units and executed synchronously on data fetched from parallel memory modules under the control of the control unit. An example of SIMD architecture is MasPar MP-2. While SIMD architectures work well for some applications, they don't provide high performance across all applications and are viewed by many as specialty hardware.

2.1.1 Vector Processors

A subclass of the SIMD systems are vector processors[2,10]. Vector processors act on array of similar data rather than on single data items using specially structured CPUs. When data can be manipulated by these vector units, results can be delivered with the rate of one, two, and in special cases three per clock cycle. So, vector processors execute on their data in an almost parallel way but only when executing in vector mode. In this case they are several times faster than when executing in conventional scalar mode. For practical purposes vector processors are therefore mostly regarded as SIMD machines.

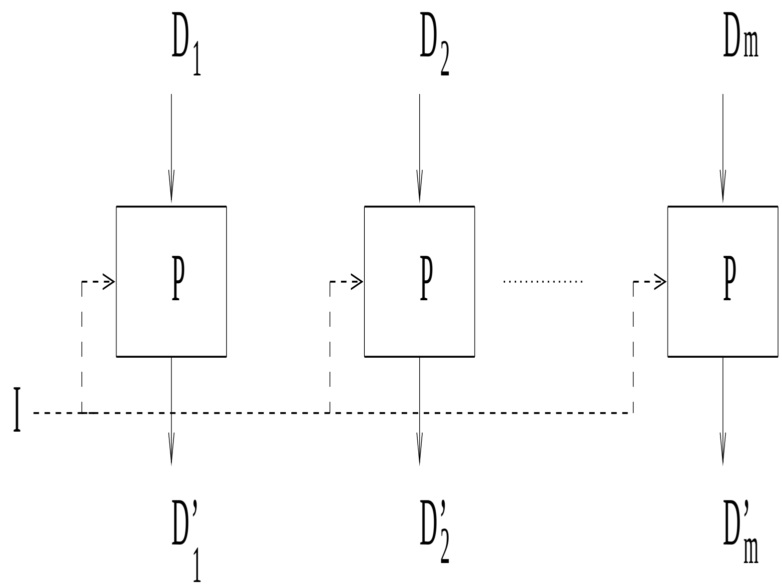

Figure 2.2: SIMD Organization

3. **Multiple Instruction Stream Single Data Stream (MISD):** $m_I > 1$, $m_D = 1$. The class of machines that can be placed in this category seems to be small. Pipeline processors may be considered to be MISD machines if the viewpoint is taken that each data item is processed by different instructions in different segments of a pipeline. This is shown in fig 2.3.

4.**Multiple Instruction Stream Multiple Data Stream (MIMD):** $m_I > 1$, $m_D > 1$. This class includes machines capable of executing several independent programs simultaneously. The class of MIMD machines appears to coincide with the class of multiprocessors. This is shown in fig 2.4. Efficient partitioning and assignments are essential for efficient multiprocessing. Most multiprocessor systems can be classified in this category.

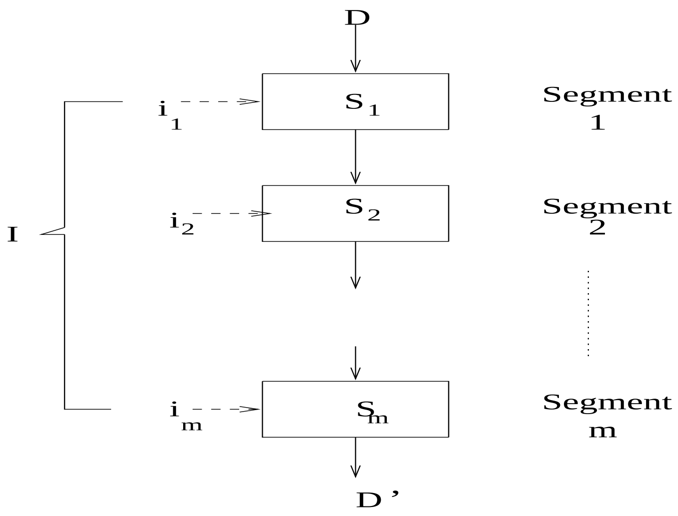

Figure 2.3: MISD Organization

2.2 Parallelism within a single processor

There are different ways to exploit parallelism with in a processor. Some of them are described below.

2.2.1 Multiple functional units

Early computers had three basic units: the main memory, the central processing unit (CPU), and the I/O subsystems. The operations were performed one function per cycle. One of the first approaches to exploiting parallelism involved splitting up the functions of ALU, having different units for the different operations and having the units operate in parallel. In order to take the advantage of the multiple functional units, the compiler had to be able to schedule operations across the multiple functional units to keep the hardware busy.

2.2.2 Pipelining

A Pipeline[2,9,10] consists of a sequences of processing circuits, called segments, through which a data stream passes. Partial processing is performed by each segment, and a final result is obtained after the data has passed through all segments of the pipeline. Occasionally it may be necessary to make several passes through the pipeline in order to process a data set completely. Parallel processing is achieved by having distinct operands sets or process in several segments at the same time. Each operand set moves from segment to segment until it has been completely processed.

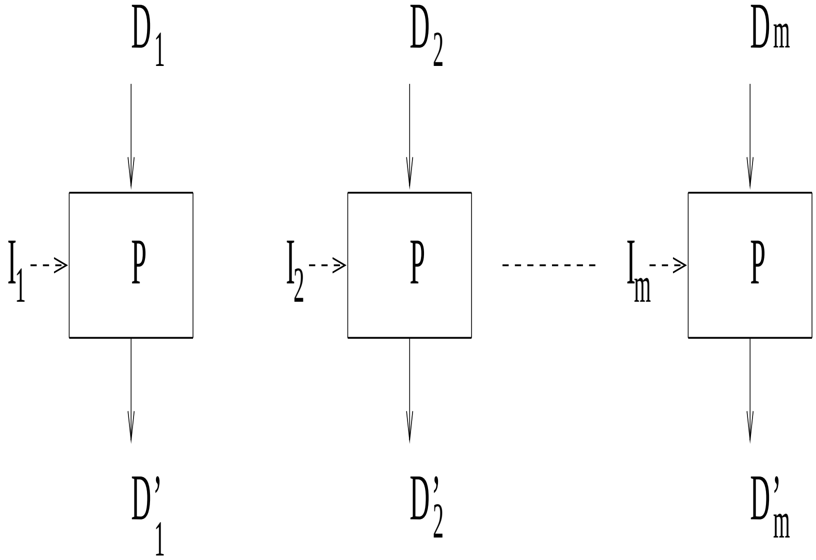

Figure 2.4: MIMD organization

Let P_1 be a non pipelined processor with total delay t_1, that is, with bandwidth t_1^{-1}. Let P_m be an m-segment pipeline version of P_1, in which each processing circuit C_i has the same delay t_C and t_R is the same delay of each segment attributable to the buffer register R_i and its associated control logic. The total delay of one segment of P_m is $t_C + t_R$; hence the maximum throughput of P_m is $(t_C + t_R)^{-1}$. Now if P_m is formed by partitioning P_1 into segments of approximately same delay, then $t_1 \approx mt_C$. Hence the maximum throughput of P_1 is approximately $(mt_C)^{-1}$. We conclude that P_m has greater maximum throughput than P_1 if the following condition is satisfied:

$$mt_C > t_C + t_R \tag{2.1}$$

There are two areas where pipelining appears to be particularly appropriate:

- The transfer of instructions through the various stages in the instruction cycle of a CPU, which amounts to realizing most or all of the CPU in the form of the pipeline called an Instruction Pipeline. The purpose of the instruction pipeline is to overlap some or all of the steps in the instruction cycle.
- The implementation of the arithmetic operations particularly the more complex arithmetic operations such as those involving floating point numbers, a pipeline processor for this purpose is called an Arithmetic Pipeline.

In the Pipeline Processors [2,9,10], pipeline scheduling takes place which allows tasks to perform without collision. The execution of the pipelined instruction incurs an overhead for filling the pipeline. Once the pipeline is filled, a result appears every clock cycle. The overhead or startup for such an operations depends on the number of stages or segments.

2.2.3 Overlapping

Some architectures allow for the overlap of operations if the two operations can be executed by the independent functional units. Overlap is similar but not identical to pipelining. Both employ the subfunction partitioning, but in a slightly different manner.

2.2.4 RISC

A RISC (Reduced Instruction Set Computer) is simpler in design and provides instruction execution at the rate of the one per clock cycle. RISC machines are often associated with pipeline implementations since pipeline techniques are natural for achieving the goals of one instruction executed per machine cycle.

There are many others architectures (e.g. VLIW) and techniques e.g. use of vector instructions, chaining, stripmining etc, used to gain the performance within a microprocessor. With these, other issues like memory organisation (in practice use of cache, main memory), data organisation, memory management came in picture and various ways have been found out and the development are still in progress. An example of how memory management should be done, is given in subsection 2.4.1.

2.3 Multi unit Processor

A multi-unit processor [2,9,10] consists of a set of m disjoint processing elements each of which is capable of acting on a data stream independently of the others. A central component of any multi unit processor system is a scheduler or control unit which coordinates activities among the various units. The scheduler must preserve the precedence among the processes being executed and the available processing units as efficiently as possible. It must keep track of the status(busy or idle) of each processing unit as well as any facilities such as registers or buses shared among them. This is actually done by associating a status bit or flag with each processing unit and shared facility. Shared memory processors come in this category. These processors are based on the MIMD organisation.

For a programming standpoint it would be attractive if a distributed memory parallel computer would allow each process to treat all of the memory in the system as a single large pool rather than separate distributed memories. This would allow the programmer to reference data not locally contained through a conventional assignment or reference rather than calls to the message passing library. This would give the illusion of shared memory in a system that has physically distributed memory. Underneath the programming layer there would be separate memories that the system would manage. This would free the programmer from writing explicit message passing statements. This type of organisation is called Virtual Shared Memory. It blurs the distinction between traditional shared memory MIMD and traditional distributed memory MIMD computing. In this memory organisation, to ensure performance, minimizing memory transfers is still critical.

2.4 System Requirements

A computer using parallel processors to achieve very high throughput poses a number of special design problems. Chief among these is that of moving instructions and data to and from the processors at or near their streaming modes. A possible solution is to partition main memory into independently addressable modules or banks so that words can be accessed simultaneously. Instructions and operands addresses should be distributed uniformly among the memory modules, so that successive access are made to different memory modules; this is called address interleaving. A communication of high bandwidth is also needed between the memory banks and the processors.

Just as instructions and data are carefully distributed among the memory modules, they must also be carefully distributed to the various processing elements. This requires the program control unit to schedule the tasks performed by the processing elements to minimize the number of conflicts and idle processors.

2.4.1 Memory Issues

A basic problem in all parallel processors with a shared main memory [1,2,10] is that of moving data and instructions between the processors and main memory at sufficiently high rates to maintain the system performance close to its maximum level. One common solution is to partition the memory into multiple modules or memory banks with address interleaving. The individual memory modules can be accessed in parallel, thereby increasing the system's effective memory bandwidth. The data and instruction address must be distributed in a balanced fashion among the available memory modules by the system software, and special hardware is needed to control access to the memory modules.

In many computer systems, large programs often cannot fit into the main memory for execution. Even if there is enough main memory for a program, the main memory may be shared between a number of users, causing any one program to occupy only a fraction of memory, which may not be sufficient for the program to execute. The usual solution is to introduce management schemes that intelligently allocate portions of memory to users as necessary for the efficient running of their programs. The goal of designing a multilevel memory hierarchy is to achieve a performance close to that of the fastest memory, at a cost per bit to that of the slowest memory. In earlier computers a technique called overlays was used, but more common today is the use of virtual memory, in which data is organized as pages and whenever a request is made, page is transfered from virtual memory to main memory, main memory to cache memory and then from cache memory to the registers inside the CPU. Performance of the computer also depends on the numbers of page faults or the number of times the data transfer takes place. Though the virtual memory management is done by the operating system and the user has no information about the pages that are actually in main memory, he can influence the number of large faults.

The example itself concerns the dense matrix-matrix multiplication [7,8] $C = AB + C$. The matrices are square and of order 1024. Assume a page contains 65,536 elements; then it follows that 64 columns of a matrix can be stored on 1 page and that all three matrices can be stored on 48 pages in total. Let us assume that the available main memory space can contain 16 pages. This implies that if we have access to successive columns of a matrix, we have a page fault after every 64 columns. Here it is also assumed that a page fault takes about 0.5 seconds of I/O time. Different ways to compute $C = AB + C$ are given below:

2.4.1.1 Inner-product approach - ijk

```
DO 30 I = 1, N
   DO 20 J = 1, N
      DO 10 K = 1, N
                 C(I,J) = A(I,K) * B(K,J) + C(I,J)
  10          CONTINUE
  20       CONTINUE
  30    CONTINUE
```

This scheme leads to paging A 1024*16*1024 times. Thus, a total number of about 16 million page faults results in a minimal I/O time of at least 93 days, and in reality the turn around time will be considerably larger since the machine usually has other works.

2.4.1.2 Column-wise update of C - jki

```
      DO 30 J = 1, N
         DO 20 K = 1, N
            DO 10 I = 1, N
              C(I,J) = A(I,K) * B(K,J) + C(I,J)
10          CONTINUE
20       CONTINUE
30    CONTINUE
```

This scheme leads to paging A 1024*16 times, B and C only 16 times. Thus, total number of about 16,000 page faults results in a minimal I/O time of about 2.25 hours.

2.4.1.3 Partitioning the matrices into block-columns - blocked jki

```
      DO 40 J = 1, N, NB
         DO 30 K = 1, N, NB
            DO 20 JJ = J, J+NB-1
               DO 10 KK = K, K+NB-1
              C(:, JJ) = A(:, KK) * B(KK, JJ) + C(:, JJ)
10             CONTINUE
20          CONTINUE
30       CONTINUE
40    CONTINUE
```

The great gain is made with respect to the I/O time: the scheme leads to paging A 4 times and B and C only once, resulting in only 96 page faults, and hence takes only a modest 70 seconds of I/O time.

2.4.2 Interconnection Issues

A parallel computer's interconnection network [10] plays a major role in determining its performance. Static interconnection structures such as trees are best suited to the execution of algorithm that requires infrequent processor communication (to share results or intermediate results etc.) and to confine most communication to neighbour processors. From the physical standpoint a network consists of a number of switching elements and interconnection links. The two major switching methodologies are circuit switching (suitable for bulk data transfer) and packet switching (suitable for short data transfer). There is one more system which is the hybrid of these two. Communication links are divided into two groups, regular and irregular. Regular are again divided into two categories: static (one dimensional, two dimensional, three dimensional, and hypercube) and dynamic (single-stage, multistage, and crossbar).

2.5 Message Passing

The message passing [10] programming model is based on the assumption that there are a number of processes in the system which have their local memories and can communicate with each other by coordinating memory transfers from one process to another. Thus, this model is of primary importance for distributed memory parallel computers. On the shared memory computers, communication between tasks executing on different processors is usually through shared data or by means of semaphores, variables used to control access to shared data. On a local-memory machine, however, communication is more explicit and is normally done through message passing.

In message passing, data is passed between processors by a single send and receive mechanism [3,10]. The main variants are whether an acknowledgement is returned to the sender, whether receiver is waiting for the message, and whether either processor can continue with other computations while the message is being sent. An important feature of message passing is that, on nearly all machines, it is much more costly than floating point arithmetic. Message passing performance is usually measured in units of time or bandwidth (bytes per second). For short messages, time is used as performance unit and for long messages bandwidth is used as a measure.

There are a number of factors that can affect message passing performance. The number of times the message has to be copied or touched (e.g. checksums) is probably the most influential effect on the message passing. Second order effects of message size may also affect performance. For small messages context switch times also contribute to delays. The other parameters, which affect the performance of the message passing, are aggregate bandwidth of the network, the amount of concurrency, reliability, scalability, and congestion management.

To enhance the development of parallel application some standard interface for parallel computing has been developed over years. Out of them MPI, PVM, and HPF are the foremost. The Message Passing Interface (MPI) [10] is a portable message passing standard that facilitates the developments of the parallel application and libraries. The standard defines the syntax and semantics of core library routines useful to a wide range of users writing portable message passing programs in Fortran, C, C++. MPI also forms a possible target for the compilers of languages such as High Performance Fortran. Commercial and free, public domain implementations of MPI already exist. These run both on tightly coupled, massively parallel machines (MPPs) and on network of workstations (NOWs). Parallel Virtual Machine or PVM [3] is used to provide an excellent platform for inter-processor communication and convert the distributed array of processors into one virtual single processor. It is a software package that allows a heterogeneous or serial computer to appear as a single concurrent computational resource. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as Ethernet, FDDI. PVM consists of two parts: a daemon process that any user can install on a machine and a user library that contains routines for initiating processes on other machines, for communicating between processes and changing the configuration of machines. PVM is available on Internet site [4] and it is a freeware tool.

2.6 Advantages of MultiProcessor System

There are several reasons to build tightly coupled systems.

1. Increase in throughput.
2. Saving of money.
3. Increase in reliability

The most common multiprocessor systems now use the *Symmetrical Multiprocessing Model*, in which each processor runs an ideal copy of the operating system, and these copies communicate with each other as needed. Some systems use *Asymmetrical Multiprocessing Model*, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

There is also a variety of reasons for building distributed systems, the major ones being these:

1. Resource sharing
2. Computation speedup
3. Reliability
4. Communication

Chapter 3

Implementation Details

Though developments in hardware enables us to exploit parallelism, but there is still a need to know about the implementation of the algorithms. Having the vision for the implementation we can decrease overhead and improve performance. In the following sections some instances will be discussed where an understanding of the implementation details [10] can be exploited to achieve better efficiency.

3.1 Parallel Decomposition and Data Dependency Graphs

Parallel computations can be described with the help of the control flow graphs [10]. Such a graph consists of nodes that represent processes and directed edges that represent execution dependencies. These graphs can represent both fine-grain and coarse-grain parallelism, but these are best suited for the coarse grain parallelism. In this case it is convenient to have the nodes represent serial subroutines or procedures that may execute in parallel. Leaves of such graphs represent processes that may execute immediately and in parallel. The graph asserts that there are no data dependencies between leaf processes. That is, there is no contention for write access to a common location between two leaf processes. The control flow representation of parallel computation differs from a data flow representation in the sense that the edges of the control flow graph do not represent data items. Instead, these edges represent the assertion that data dependencies do not exist once there are no incoming edges. The programmer must take the responsibility for ensuring that the dependencies represented by the graph are valid. For an example, the term execution dependencies means that one process is dependent upon the completion of, or output from, another process and is unable to begin execution until this condition is satisfied. Additional processes may be dependent on this process.

A special case of the execution dependency is deadlock [2,10]. This phenomenon, encountered only in MIMD machines, is caused by a circular dependency chain so that all activity ceases, with all processors either idle or in a wait state. This is the most difficult bug to detect or fix. In particular, acyclic control flow graphs don't admit deadlocks. Thus, describing a parallel decomposition in terms of such a graph avoids this problem from the outset. One example is shown in figure 3.1. This abstract representation of the parallel decomposition graph can easily be reformed into a parallel program. A correct understanding of the data dependencies among communicating parallel processes is, of course, essential to obtain a correct parallel program. Once these dependencies have been understood, a control flow graph can be constructed that will ensure that the data dependencies are satisfied. A control flow graph may go further by introducing execution dependencies to force a particular order of computation if that is deemed necessary for the purpose of load balancing or performance.

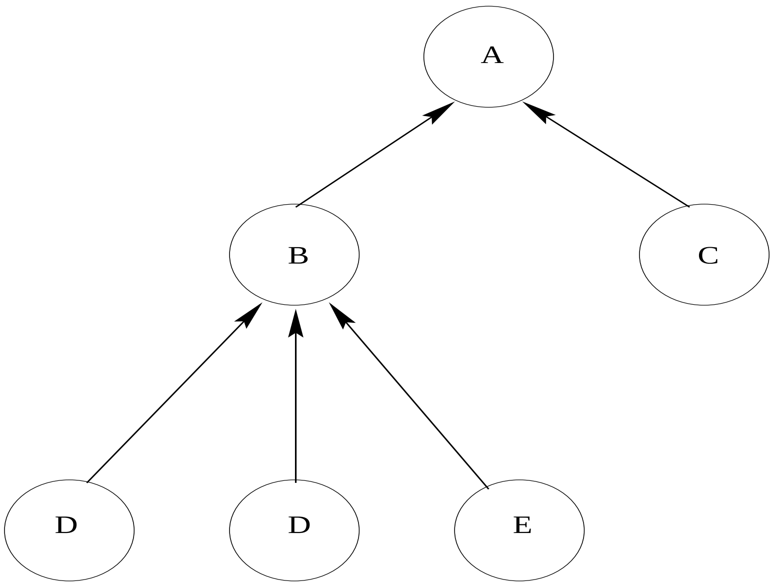

Figure 3.1: Large-Grain control flow graph

3.2 Synchronization

Synchronization [1,2,10] is required when two or more concurrently executing process must communicate, to signal their completion to each other, or simply to provide information to the other of its current progress. The crudest and most common form of synchronization is that of the barrier often used in the join of a fork-and-join. Here a task records its completion and halts. Only when all of the processes have reported to the barrier, the task halted at the barrier can continue further.

Most shared memory processors use lock function [1] for the synchronization. A lock function tests the value of the lock variable. If the value is on, the process is blocked until the lock variable is off. When the lock variable is found to be off, it is read and set to on in an autonomous operation. Thus, exactly one process may acquire a lock variable and set it to on. A companion to a lock function is an unlock function, which serves to set the lock variable to off. When a task wishes to have exclusive use of a section of the code (or exclusive use of data accessed), it tries to set the lock through a subroutine call. In this situation a programmer has a section of program that is to be executed by at most one process at a time, called a critical section. If the lock has already been set by another task, it is unable to set the lock and continue into the critical section, until the task, that sets a particular lock unsets or unlocks it. An example of the locks, unlocks and critical section is given below:

```
CALL LOCK( <lock variable> )
       execute critical section
CALL UNLOCK( <lock variable> )
```

A more flexible and powerful form of synchronization can be achieved through the use of *events* [1]. Events can be posted, awaited, checked for, and unposted. Any processor can unpost an event, and after checking its existence, the programmer is free to continue if appropriate. This form of synchronization is becoming increasingly common.

3.3 Load balancing

Load balancing [1,10] involves allocating tasks to processors so as to ensure the most efficient use of resources. With the advent of parallel processing, the ability to make iterations of loops run in parallel, refered to as microtasking, has become increasingly important. It can be very efficient to split off even small tasks: the determining factor is whether wall clock time has been reduced without having to pay significantly more for the increase in CPU time (resulting from parallel overhead).

When a job is multiprocessed, a common measure of efficiency is the ratio of the elapsed time on one processor to that on p processors divided by the p. This ratio reflects both time lost in communication or synchronization and idle time on any of the p processors. It is the idle time that load balancing strategies seek to minimize.

For an example, to illustrate the effect of load balancing, let us assume that there are four processors and seven independent tasks, which simultaneously make one work. Let one of the tasks requires 2 units of time and rest takes 1 unit of time each. Then, an optimized distribution of the tasks will be such that three processors get six tasks of 1 unit of time, each having two tasks and fourth one has the task of 2 unit time. In this way the work will be completed in the 2 unit of time. Any other partition of tasks will take more time than the previous partition.

Suppose we take a system, that is multiprogrammed, if one or more processors are idle, that should not be of great concern since the idle time can be taken by another job. This point is of particular importance when the parallelism, measured in the number of simultaneously executable tasks, varies during the course of a job. Thus it might be efficient at one time to use all the processors of a system while at another time to use only one or two processors. Finally, it is observed that on a highly parallel machine, many tasks are required to saturate the machine, so that the problem size has to be large enough to generate enough parallelism.

Chapter 4

Performance Analysis

4.1 Performance Consideration

The performance [2,10] of a parallel processor depends on complex and hard to define ways on the system's architecture and degree of parallelism in programs it executes. There are several simple performance measures such as speedup and processor utilization (efficiency) which provides rough performance estimates. The system interconnection structure and main-memory architecture also have a significant impact on performance.

4.1.1 Basic Measures

Here some basic terms are defined which are used to evaluates the performance of parallel processors.

4.1.1.1 Parallelism Degree

Individual or average instruction execution times are not very useful, since they don't measure the impact of what might be loosely called the system's parallelism degree n [2], e.g. the number of processors or the number of pipeline segments available. Sometimes especially in theoretical studies, the time T(n, N) needed to execute more complex operations such as sorting a list of N numbers, or inverting an $N * N$ matrix, is given as a performance measure for a machine of parallelism n. Such measures may be expressed in approximate form using the $\mathcal{O}$-*notation* $\mathcal{O}(\ f(n, N))$, where stating that $\mathrm{T}(n,N) = \mathcal{O}(\ f(n, N))$ means that there exist constants k and $N_0 > 0$, whose exact values are not usually specified, such that $T(n, N) \leq kf(n, N)$ for all $N > N_0$. As N, the problem size or dimension increases beyond some lower limit N_0, the execution time $\mathrm{T}(N)$ grows at a rate that is less, within a constant k, than that of the function $f(n, N)$.

The performance of an individual processor or processing elements in high performance machines may be measured by the instruction rate or instruction bandwidth b_l, using units MIPS. The corresponding measures of data bandwidth or throughput b_d is typically MflopsS; other multiple such as GFLOPS are also used. In parallel computers the interprocessor communication mechanism, and the extent to which they are used, strongly influences overall system performance. For example for a system of n processors, each with performance p, maximum or potential performance turns out to be n*p. Performance figure like this is obtained for highly parallel programs or algorithms, where the maximum possible parallelism in computation is attained, and communication delay is minimized. Under most typical operating conditions, when a mixture of tasks with varying degree of parallelism are encountered, significant lower bounds can be expected.

4.1.1.2 Speedup

Speedup $Sp(n)$ [2,10], is defined as the ratio of the total execution time $T(l)$ on a sequential computer to the corresponding execution time $T(n)$ on the parallel computer for some class of tasks of interest.

$$Sp(n) = \frac{T(l)}{T(n)} \tag{4.1}$$

It is usually reasonable to assume that $T(l) \leq n * T(n)$, in which case $Sp(n) \leq n$.

4.1.1.3 Efficiency

A closely related performance measure, which can be expressed as a single number (a fraction or a percentage), is the efficiency E_n [2,10], which is the speedup per degree of parallelism, and is defined as follows:

$$E(n) = \frac{S(n)}{n} \tag{4.2}$$

$E(n)$ is also an indication of processor *utilization*, and may be so named.

In general, speed up and efficiency provide rough estimates of the performance changes that can be expected in a parallel processing system by increasing the parallelism degree n, e.g. by adding more processors. These measures should be used with caution, however, since they depend on the program being run, and can change dramatically from program to program, or from one part of the program to another.

4.2 Amdahl's Law

A program or an algorithm Q may sometimes be characterized by its degree of parallelism n_i, which is the minimum value of n for which the efficiency and speedup reach their maximum values at time i during the execution of Q. Thus n_i represents the maximum level of parallelism that can be exploited by Q at time i.

Some indication of the influence of program parallelism on the performance of a parallel computer may be seen from the following analysis. Suppose that all computations of interest on a parallel processor can be divided into two simple groups involving floating point arithmetic only: vector operations employing vector operands of some fixed length N, and scalar operations where all operands are scalars(N=1). Let $1-f$ be the fraction of all floating point operations that are executed as scalar operations, and let f be the fractions executed as vector operations. f is thus a measure of the degree of parallelism in the program being executed, and varies from 1, corresponding to maximum parallelism (all vector operations), to 0 (all scalar operations). Suppose that vector and scalar operations are performed at throughput rates of V and S Mflops, respectively. Let the total CPU time be t . Let a given algorithm has N flops. Then, relation of f, N, V, S, t are given by the following useful formula:

$$t = \frac{(1-f)N}{S} + \frac{fN}{V} \tag{4.3}$$

Equation 4.3 can be used to evaluated the performance of parallel processors in the context of vector and scalar operations. It is found out that the parallel processor is very sensitive to the scalar operations. A very small fraction of scalar operations drops the system performance by a significant amount. Hence it is worthwhile to devote considerable effort to "vectorize" or "parallelize" programs for such machines to eliminate scalar operations. With $1-f$ interpreted broadly as the fraction of nonparallelizable operations or instructions. This equation is often referred as ***Amdahl's Law*** [2,10]. If the algorithm had been executed completely with the lower speed S, the CPU time would have been $t = \frac{N}{S}$, where N is the number of operations algorithm involves. This implies that the relative gain in the CPU time obtained from executing the portion fN at the speed V instead of the much lower speed S is bounded by $\frac{1}{1-f}$. Thus, f must be rather close to 1 in order to benefit significantly from high computational speed.

4.2.1 General form of Amdahl's Law

Assume that the execution of an algorithm consists of executing the consecutive parts $A_1, A_2, ..., A_n$ such that the N_j flops of part A_j are executed at a speed of r_j Mflops. Then the overall performance r [10] for the $N = N_1 + N_2 + ... + N_n$ flops is given by

$$r = \frac{N}{\frac{N_1}{r_1} + \frac{N_2}{r_2} + ... + \frac{N_n}{r_n}} Mflops \tag{4.4}$$

This follows directly from the observation that the CPU time t_j for the execution of part A_j is given by $t_j = \frac{N_j}{r_j}$ microseconds.

From Amdahl's law we can study the effect of different speeds for separate parts of a computational task. It is seen that the improvements in the execution of computations usually are obtained only after the startup time. This fact implies that the computational speed depends strongly on the amount (and type) of the work.

4.2.2 Amdahl's Law for Pipeline Processors

Pipeline processors give result after startup time. Once the pipeline is filled the results come in comparatively shorter duration. For large vectors computing speed will be large compared to the short vectors, where it is barely noticeable. The performance r_N for a given loop of length can be formulated in terms of r_∞ and $n_{1/2}$ [10]. The values of these parameters are characteristic for a specific loop; they represent the performance in Mflops for very long loop length (r_∞), and the loop length for which the performance of about $1/2 r_\infty$ Mflops is achieved ($n_{1/2}$). For many situations we have fairly good approximation that

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \tag{4.5}$$

According to 4.5, for a loop length $N = qn_{1/2}$ a performance of r_∞q/(1+q) Mflops will be obtained. Consequently, it is highly desirable that values of $n_{1/2}$ should be small relative to N. As a rule of thumb $n_{1/2}$ and r_∞ are proportional to the number of pipelines of a given computer. For larger number of pipelines the value of r_∞ will be larger. For a p-processor model system this approach will lead to an increase of r_∞ and $n_{1/2}$ by a factor of p relative to the single processor execution for multiple loops. In general, on a p processor model r_∞ grows less than with p, and $n_{1/2}$ grows faster, because of synchronization effect.

4.2.3 Amdahl's Law for Parallel Processing

4.2.3.1 A simple model

Let t_j denote the wall clock time to execute a given job on j parallel processors. The speedup, S_p, for a system of p processors is then by definition

$$S_p = \frac{t_1}{t_p} \tag{4.6}$$

The efficiency, E_P of the p processor model is given by,

$$E_p = \frac{S_p}{p} \tag{4.7}$$

It can be seen that $0 \leq E_p \leq 1$ Let us assume for simplicity that a job consists of basic operations all carried out with the same computational speed and that a fraction f of these operations can be carried out in parallel on p processors, while the rest of the work can keep only one processor busy. The wall clock time for the parallel part is then given by $\frac{ft_1}{p}$; and , if we ignore all synchronization overhead, the time for the serial part is given by $(1-f)t_1$. Consequently, the total time for the $p - processor$ model is given by:

$$t_p = \frac{ft_1}{p} + (1-f)t_1 = \frac{t_1(f+(1-f)p)}{p} \geq (1-f)t_1 \tag{4.8}$$

From this equation speedup S_p can be calculated directly:

$$S_p = \frac{p}{(f+(1-f)p)} \tag{4.9}$$

For $f < 1$ the speedup S_p is bounded by $S_p \leq \frac{1}{1-f}$. This equation 4.9 is known as **Ware's law**[10]. This equation is devised having assumption of an idealized system. For practical purpose, it will be more complicated.

In most practical situations it is quite unrealistic to assume that a given job consists of only two parts, a strictly serial part and one that can execute in parallel on p processors. Often, some parts can be executed in parallel on p processors while other parts can still be executed in parallel but on fewer than p processors. It is seen that an increase in the number of processors usually goes along with an increase in the problem size. Let the non-parallel part of the computations be denoted by $\alpha M = 1 - f$, where M is a measure of the total number of operations. For any algorithm, $\alpha(M)$ decreases when M is increased. Thus, an "effective parallel algorithm" is defined as an algorithm for which $\alpha(M)$ goes to zero when M grows. If the parallel part can be executed in parallel on q processors, then it follows for such an algorithm that

$$S_p = \frac{q}{1 + (q-1)\alpha(M)} \tag{4.10}$$

$$E_p = \frac{1}{1 + (q-1)\alpha(M)} \tag{4.11}$$

These two equations are known as **Moler's equations**[10], hence S_q grows to q and E_q grows to 1 for growing M. If local memory is sufficiently large enough, then S_p will be close to q. In many situations memory size is a bottleneck in realizing large speedups, simply because it often limits the problem size.

4.2.3.2 Gustafson's Model

The basic premise of the Amdahl's law is that the proportion of code that can be vectorized or parallelized remains constant as vector length that is, problem size increases. An alternative model, Gustafson's model [10], can be based on fixing the computation time rather than the problem size, as f varies. Thus, in Gustafson's model, if we assume that the problem can be solved in an unit time on our parallel machine, then the time in serial computation is $1-f$ and in parallel computation is f, then if there are p processors in our parallel system, the time for a uniprocessor to perform the same job would be $1-f+fp$. Then the speedup, $S_{p,f}$, is given by:

$$S_{p,f} = p + (1-p)(1-f) \tag{4.12}$$

As the number of processors increases, so does the proportion of the work in parallel mode, and speedup is not limited in the same way as in Amdahl's law.

Chapter 5

Linear Algorithms

One of the most important aspects to utilizing a high-performance computer effectively is to avoid unnecessary memory references. In most computers data flows from memory to registers and from registers to functional units, which perform the given instructions on the data. Performance [10] of the algorithms [6] can be dominated by the amount of memory traffic, rather than the floating point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on the data. This cost provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement.

5.1 Basic Factorizations of Linear Equations

Overall system matrix [7,8,10] of many scientific and technical problems turns out to be in the form, $Ax = b$. Here A is a real nxn matrix and b and x are the x both real vectors of length n. A may be dense or sparse matrix. Here we will develop algorithm for the dense coefficient matrix. Equivalently, it can be said that the solution of the linear system, given in equation 5.1, has to be found.

$$Ax = b \tag{5.1}$$

To construct a numerical algorithm, we rely on the fundamental result from linear algebra that implies that a permutation matrix exists such that

$$PA = LU \tag{5.2}$$

where L is the unit lower triangular matrix (ones on the diagonal) and U is upper triangular. The matrix U is non-singular if and only if the original coefficient matrix A is non-singular. This factorization facilitates the solution of equation 5.1 through the successive solution of triangular systems

$$Ly = Pb, Ux = y \tag{5.3}$$

The well known numerical technique for constructing this factorization is called Gaussian elimination [8,10]] with partial pivoting. A brief development of this basic factorization is given in the next section. Variants of this factorization will be covered in the successive sections.

5.2 Point Algorithm

This is also known as Gaussian elimination with partial pivoting [10]. Let P_1 be a permutation matrix such that

$$P_1 A e_1 = \begin{bmatrix} \delta \\ c \end{bmatrix} \tag{5.4}$$

where is $|\delta| \geq |c^T e_j|$ for all j (i.e., δ is an element of largest magnitude in the first column). If $\delta \neq 0$, put $l = c\delta^{-1}$; otherwise put $l = 0$. Then

$$P_1 A = \begin{bmatrix} \delta & u^T \\ c & \hat{A} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l & I \end{bmatrix} \begin{bmatrix} \delta & u^T \\ 0 & \hat{A} - lu^T \end{bmatrix} \tag{5.5}$$

Suppose now that $\hat{L}\hat{U} = \hat{P}(\hat{A} - lu^T)$, then

$$\begin{bmatrix} 1 & 0 \\ 0 & \hat{p} \end{bmatrix} P_1 A = \begin{bmatrix} 1 & 0 \\ \hat{P}l & \hat{L} \end{bmatrix} \begin{bmatrix} \delta & u^T \\ 0 & \hat{U} \end{bmatrix} \tag{5.6}$$

and $PA = LU$. This discussion provides the basis for an inductive construction of the factorization $PA = LU$. Now, repeat the basic factorization steps on the "reduced matrix" $\hat{A} - lu^T$ and continuing in this way until the final factorization has been achieved in the form

$$A = (P_1^T L_1 P_2^T L_2 P_3^T L_3.....P_{n-1}^T L_{n-1})U \tag{5.7}$$

with each P_i representing a permutation in the (i, k_i) position where $k_i \geq i$. This representation leads to the following numerical procedure for the solution of equation 5.1.

for $j = 1$ step 1 until n

$b \leftarrow L_j^{-1} P_j b;$

end;

Solve $Ux = b$;

5.3 Blocked Algorithms

Many algorithm didn't perform near the expected level on the powerful machines. The key to achieving high performance on these advance architectures has been to recast the algorithm in terms of matrix-vector [7,10] and matrix-matrix operations to permit reuse of data. To derive these variants, a method block factorization will be studied. One can construct the factorization by analyzing the way in which the various pieces of the factorization interact. Let us consider the decomposition of the matrix A into its LU factorization [8] with the matrix partitioned in the following way. Let us suppose that we have factored A as $A = LU$. This can be written in the following form, where the factors are in block form.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \tag{5.8}$$

Multiplying L and U together and equating terms with A, the following will result:

$$\begin{array}{lll} A_{11} = L_{11}U_{11}, & A_{12} = L_{11}U_{12}, & A_{13} = L_{11}U_{13} \\ A_{21} = L_{21}U_{11}, & A_{22} = L_{21}U_{12} + L_{22}U_{22}, & A_{23} = L_{21}U_{13} + L_{22}U_{23} \\ A_{31} = L_{31}U_{11}, & A_{32} = L_{11}U_{12} + L_{32}U_{22}, & A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \end{array} \tag{5.9}$$

With these simple relationships, variants can be developed by postponing the formation of certain components and also by manipulating the order in which they are formed. A crucial factor for performance is the choice of the blocksize, k, i.e. column width, of the second block column. A blocksize of 1 will produce matrix-vector algorithm and for $k > 1$ matrix-matrix algorithms will be produced. Machine dependent parameters such as cache size, number of vector registers, and memory bandwidth will dictate the best choice for the blocksize.

Three natural variants occur: right-looking [10], left-looking[10], and hybrid of these two, named as Crout [10]. The left-looking variants computes one block column at a time using previously computed columns. The right-looking variant computes a block row and column at each step and uses them to update the trailing sub-matrix. The Crout variant is a hybrid algorithm in which a block row and column are computed at each step and using previously computed rows and previously computed columns. These variants have been called the i, j, k variants owing to the arrangements of loops in the algorithm. The terms right and left refer to the regions of data access.

5.3.1 Right-Looking algorithm

Suppose that a partial factorization of A has been obtained so that the first k columns of L and the first k rows of U have been evaluated. The factorization will be in the form of:
STEP 1:

$$PA = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & \hat{A_{22}} & \hat{A_{23}} \\ 0 & \hat{A_{32}} & \hat{A_{33}} \end{bmatrix} \tag{5.10}$$

where L_{11} and U_{11} are kxk matrices and P is a permutation matrix representing the effects of pivoting. The blocks labeled $\hat{A_{ij}}$ are the updated portion of A and has not yet been factored, and will be referred to as the active sub-matrix.
STEP 2:

$$\begin{bmatrix} I & 0 \\ 0 & P_2 \end{bmatrix} PA = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & \hat{A_{33}} \end{bmatrix} \tag{5.11}$$

where P_2 is the permutation matrix of the order $n - k$.
STEP 3:

$$P_2 \begin{bmatrix} \hat{A_{22}} \\ \hat{A_{32}} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22} \tag{5.12}$$

STEP 4:

$$\begin{bmatrix} \hat{A_{22}} \\ \hat{A_{32}} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} \hat{A_{22}} \\ \hat{A_{32}} \end{bmatrix} \tag{5.13}$$

and

$$\begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \tag{5.14}$$

STEP 5: Solve the triangular system

$$U_{23} = L_{22}^{-1} \hat{A_{23}} \tag{5.15}$$

to compute the next block row of U.
STEP 6: Update $\hat{A_{33}}$

$$\hat{A_{33}} \leftarrow \hat{A_{33}} - L_{32}U_{23} \tag{5.16}$$

STEP 6: *repeat* the STEP 2 - STEP 6, *until* $A = LU$

The main advantage of the block partitioned form of the LU factorization algorithm is that the updating of $\hat{A_{33}}$ involves a matrix-matrix operation if the block size is greater than 1. Matrix-Matrix operations generally perform more efficiently than matrix-vector operations on a high performance computer. The original array A may be used to store the factorization, since L is unit lower triangular and U is upper triangular. The additional zeros and ones appearing in the representation do not need to be stored explicitly.

5.3.2 Left-Looking algorithm

From the standpoint of data access this is the best algorithm of the three.
STEP 1: Assume that

$$PA = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} U_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & A_{32} & A_{33} \end{bmatrix} \tag{5.17}$$

STEP 2:

$$\begin{bmatrix} I & 0 \\ 0 & P_2 \end{bmatrix} PA = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & A_{13} \\ 0 & U_{22} & A_{23} \\ 0 & 0 & A_{33} \end{bmatrix} \tag{5.18}$$

STEP 3: Solve the triangular system

$$U_{12} = L_{11}^{-1} A_{12} \tag{5.19}$$

STEP 4: Update the rest of the middle block column of U,

$$\begin{bmatrix} \hat{A_{22}} \\ \hat{A_{32}} \end{bmatrix} \leftarrow \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} - \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} U_{12} \tag{5.20}$$

STEP 5: Perform the factorization

$$P_2 \begin{bmatrix} \hat{A}_{22} \\ \hat{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22} \tag{5.21}$$

STEP 6: Lastly the pivoting

$$\begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} \tag{5.22}$$

and

$$\begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \tag{5.23}$$

It can be seen that the data accesses all occur to the left of the block column being updated. Moreover, the only write access occurs within this block column. Matrix elements to the right are referenced only for pivoting purposes, and even this procedure may be postponed until needed with a simple rearrangement of the above operations. The original array A may be used to store the factorization, since L is unit lower triangular and U is upper triangular. The additional zeros and ones appearing in the representation do not need to be stored explicitly.

5.3.3 Crout Algorithm

The Crout variant is best suited for vector machines with enough memory bandwidth to support the maximum computational rate of the vector units. This algorithm requires fewer memory references than the right-looking algorithm.

STEP 1: Assume that

$$PA = \begin{bmatrix} L_{11} & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & U_{12} & U_{13} \\ L_{21} & A_{22} & A_{23} \\ L_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} U_{11} & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \tag{5.24}$$

STEP 2: For factorization,solve

$$\begin{bmatrix} A_{22} & A_{23} \end{bmatrix} \leftarrow \begin{bmatrix} A_{22} & A_{23} \end{bmatrix} - L_{12} \begin{bmatrix} U_{12} & U_{13} \end{bmatrix} \tag{5.25}$$

and

$$A_{32} \leftarrow A_{32} - L_{31}U_{12} \tag{5.26}$$

STEP 3: Do factorization

$$P_2 \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22} \tag{5.27}$$

STEP 4: Replace

$$\begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} A_{23} \\ A_{33} \end{bmatrix} \tag{5.28}$$

and

$$\begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \leftarrow P_2 \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} \tag{5.29}$$

and put

$$U_{23} = L_{21}^{-1} A_{23} \tag{5.30}$$

STEP 5:

$$P_2 PA = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & 0 & U_{13} \\ 0 & I & U_{23} \\ L_{31} & L_{32} & A_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & 0 \\ 0 & U_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \tag{5.31}$$

STEP 6: *repeat* STEP 2 - STEP 5 *until* LU decomposition complete.

The lower triangular factor L_{11} and the upper triangular factor U_{11} may occupy the left-hand corner of the original nxn array. Thus, the original array A may be overwritten with this factorization as it is computed.

Chapter 6

Future Work

We discussed various aspects for the parallel processing and high performance computing in the report. The future work will involve the following:

1. A detailed survey of computer architecture, specifically for pipelined processors,is to be done.

2. Since optimisation of the code is of utmost importance, so the in-depth knowledge is to be gained in the field of compiler theory.

3. Message passing strategies need to be explored in a greater detail.

4. Linear Algebra algorithms for the uniprocessor computation and its extensions for the multiprocessor paradigm are to be devised.

5. There is a need for the general algorithms to be made compatible to the multiprocessor systems.

6. Algorithms have to be discovered or divised, which are best suitable for shared memory systems.

Acknowledgment

I would like to express my deep sense of gratitude to **Prof. H. Narayanan** and **Prof. Madhav P. Desai**, for their invaluable help and guidance during the course of project. I am highly indebted to them for constantly encouraging me by giving their critics on my work. I am grateful to them for having given me the support and confidence.

Gaurav Trivedi
May 2000
Indian Institute of Technology, Bombay

References

[1] Abraham Silberschatz, Peter Baer Galvin, "Operating System Concept", Addison Wesley, Reading Massachusetts, USA, 1998

[2] John P. Hayes, "Computer Architecture and Organization", McGraw-Hill International Company, Singapore, 1988

[3] PVM 3 User Guide and Reference Manual, Edited by Al Gist, Oak Ridge National Laboratory, Engineering Physics and Mathematics Divison, Mathematical Science Section, Oak Ridge, Tennessee, USA, 1991

[4] PVM's HTTP Site, "http://www.epm.ornl.gov/pvm/"

[5] Brian W. Kernighan, Dennis M. Ritchie, "The C - Programming Language, (ANSI C Version)", Prentice-Hall of India Pvt. Ltd., New Delhi, 1998

[6] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithm", MIT Press, Cambridge, MA, USA, 1990

[7] Kenneth Hoffmann, Rey Kunze, " Linear Algebra", Prentice-Hall of India Pvt. Ltd., New Delhi, 1997

[8] G.H. Golub and C. F. Van Loan , " Matrix Computations", Third Edition. The Johns Hopkins University Press, Baltimore, 1996

[9] David A. Patterson, John L. Hennessy, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publications Inc., San Mateo, California, USA, 1990

[10] Jack Dongarra, Iain Duff, Danny Sorensen, and Henk van der Vorst, Numerical Linear Algebra for High-Performance Computing",Society for Industrial and Applied Mathematics, Philadelphia, 1998