UNIT – I Parallel Processing Concepts

1.1 Introduction to Parallel Computing

- Parallel Computing is typically used to describe about the solving of a single problem using two or more processors. In the current developments the use of multiple cores also termed as parallel computing.
- A simple example will better state about it, suppose we have to perform arithmetic operations and we have quad core machine.
- The program is divided in four functions for obvious reasons and each function will be assigned into individual cores. Once the inputs required to perform the operations specified in the functions become available all functions execute independently in individual core and computations will be carried out in less amount of time.
- It becomes a need to compare parallel computing with serial computing because we are all habitual to do serial computing using the computers. One question always arises in our mind about parallel computing is whether it is harder than serial computing or not?
- Parallel computations can be performed only when interdependencies amongst operations will not take any significant time.
- The programmer is responsible to sort out the details about these and also need to deal with the communications required among the operations implemented in different functions.
- As we all know the largest and most powerful computers are called supercomputers. In fact supercomputers are highly parallel machined equipped with number of processors to work on the same problem.
- It is always possible that the parallelism can appear at different levels and that makes it difficult to define precisely.
- One of the levels parallelisms is the instruction level parallelism in which, within a CPU several instructions work simultaneously.
- In instruction level parallelism a compiler derives the instructions and processor decides which instructions can be processed simultaneously. Another form of parallelism called as task level parallelism is based upon the fact that multiple CPU's are assigned to process multiple instruction streams simultaneously.

1.2 Motivation for Parallelism

- There are many factors included behind the use of parallel computing in the current trend, some of the significant factors will be mentioned in this season.
- Continuing in the topic of parallel computing introduction we will again describe parallel computing in brief with illusion of a pictorial representation. Parallel computing is used to refer the assignment of multiple processors simultaneously to solve a computational problem.

- The Fig 1.2.1 explains using pictorial representation the task level parallel computing. Here the overall problem to be solved is represented by a big rectangular shape.
- The problem is divided into four subtasks and each is represented by four smaller rectangular shapes.
- Now by the use of four CPU's each part of the sub problems is processed simultaneously. In this way all subtasks of a problem are computed together in different CPUs. In this way the computation time is reduced significantly.

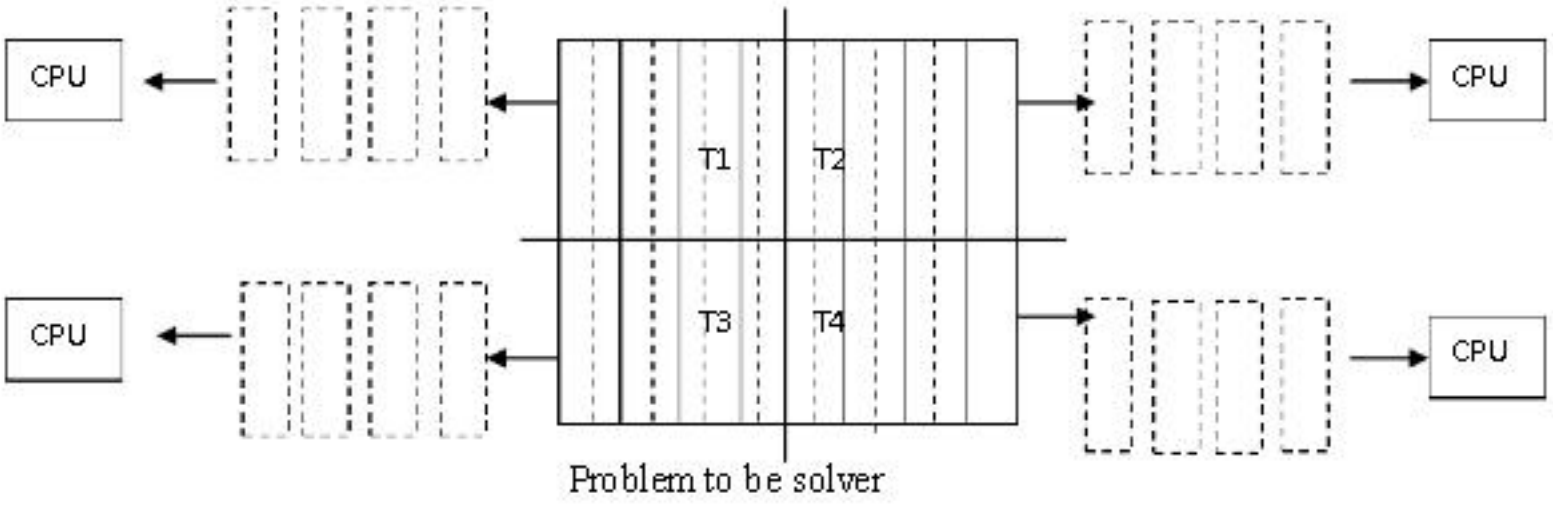


Fig. 1.2.1: Parallel computation of tasks

- The most influential factor for the adaptation of parallel computing is the availability of powerful hardware devices.
- We will discuss various reasons directly or indirectly affects the utilization of parallel computing so that the actual facts behind the use of parallel computing will be explored.

1.2.1 Increased Number of Transistors in Integrated Chips Enhances Computing Power

- The power of CPUs increases because of the additional transistors in the ICs. The overall computing power improvements in the modern processors opens the doors for parallel utilization of processing elements.
- The modern processing elements are equipments with the multiple units and these units can be assigned to perform processing tasks individually.
- According to Moore's Law the number of transistors would double in every two years. The counts of transistors included in devices are very large but the use of all the transistors must be feasible then only the collective goal to achieve fast processing capability will be met.
- The parallel processing techniques are applied to utilize these devices effectively. The two forms of parallel processing implicit and explicit can be used with these devices. In later chapters these forms of parallelism are discussed in detail.

1.2.2 Improvements in Storage Technology (Memory/Disk)

- As we all know that all modern computers work on the basis of Von Neumann architecture. According to this approach the program and data stored in the permanent storage and during executions should be loaded in the primary memory.
- This makes clear that processor alone is not responsible for the enhancement in computations speed but primary memory as well as disk speed also plays very significant roles.
- The access time of DRAM has increased but not yet increased much as compare to processor clock rates. This showed major performance bottleneck between a processor and a memory unit.
- We further elaborate this as the number of instructions executed in per clock cycle by the processor has increased but at the same time memory should be able to feed the required data for execution.
- The technique called as locality of reference is used to manage the mismatch between the memory and the processor speed.
- In fact additional faster memory termed as cache memory is used to cope up with the situation.

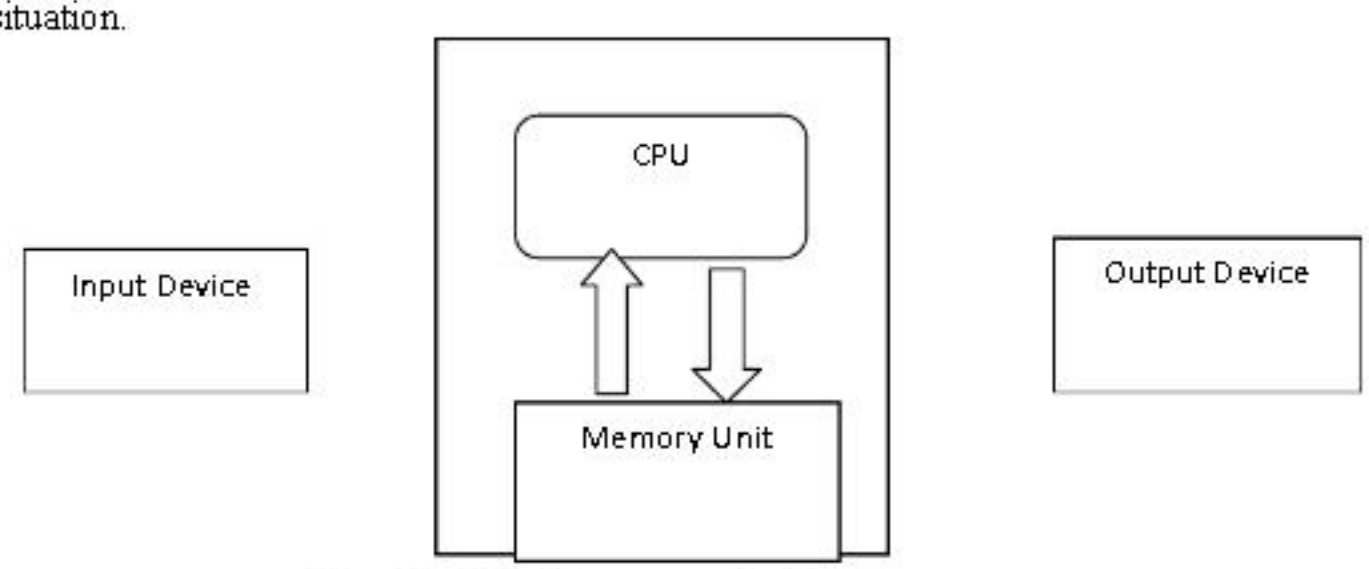


Fig. 1.2.2 Von Neumann Architecture

Fig. 1.2.3 Cache and main memory

- The factors counted in representation of the memory performance are the speed and the latency. In fact the relationship between speed and latency is used to show the performance of the memory system.

- The memory latency is the delays occur in transmitting the data from RAM to the processor.
- The collective performance of the memory system is based upon the overall memory requests handle from the cache memory.
- Now when we focus about the caches and bandwidth in parallel computing platforms it is obvious that the memory system performance is better. In parallel systems because of the existence of multiple processors the overall cache size becomes larger.
- In the similar fashion the larger disk capacity is also be achieved for the permanent storage purposes.
- The efficient utilization of parallel systems can be achieved while running parallel algorithms. The performance of such algorithms depend upon the speed at which data transferred to and from the storage systems instead of fully relying on computations performed by the processors.

1.2.2 Improved Networking Devices and Systems for Data Communications

- The recent trend in computing is the use of Internet as a large platform for parallel and distributed computations. This means Internet is used to provide the environment for performing large computations.
- The inventions of much software systems and tool for such a platform started a new field for application developments.
- Many parallel applications suits for Internet based distributed computing platforms. In all such applications the need for travelling of data from one node to other nodes effectively handled by several networking devices. This is made possible because of the growth in the networking devices.
- This is made possible because of the growth in the networking devices and technologies.

1.3 Scope of Parallel Computing

- The parallel computing is suitable for the problems require much more time for computational completion. The several areas in which parallel computing are used usually based on high – end engineering and scientific problems.
- This includes computer based simulations and Computational Fluid Dynamics (CFD) as well as other computer based digital image processing and security algorithms. In this section we will cover brief descriptions about such areas.

1.3.1 Applications of Parallel Computing in Engineering

- Parallel computing is used in various engineering problems. These problems typically solved in the form of computer programs in traditional machines.
- These machines are capable of performing the tasks sequentially but designers closely observed and found the subtasks included in the problem can be performed as different processes simultaneously.
- The parallel computations provide the improvements in overall efficiency of the application. The main factor responsible for efficiency improvements is the speedup of the computations.

- This phenomenon is adopted in several engineering application domains such as application of aerodynamics, optimization algorithms like branch and bound and genetic programming etc.

1.3.2 Scientific Applications using Parallel Computing

- In short scientific applications are created to show the simulated behaviors of rest world entities by using mathematics and mathematical formulas.
- This means the object exists in the real world are mapped in the form of mathematical models and the actions present in that objects are simulated by executing the formula.
- Simultaneous are based on very high – end calculations and require the use of powerful computers. The most of the powerful computers are parallel computers and do the computations in parallel forms.
- The scientific applications are major candidates for the parallel computations. For example weather forecasting and climate modeling, Oil exploration and energy research, Drug discovery and genomic research etc. are some of the scientific applications in which parallel activities are carried out throughout the completion
- These activities are performed by the tools and techniques provided under the field of parallel computing.

1.3.3 Commercial Applications Based on Parallel Computing

- The increasing need for more processing power is full-filled by parallel architectures. The parallel architectures continuously evolve to satisfy the processing requirements of the developing trends of the society.
- The commercial applications require more processing power in the current market trends because of performing many activities simultaneously.
- This creates an inclination towards the use of parallel computing application for commercial problems. We can consider multimedia applications as an example of commercial applications in which parallel computing plays vital role.
- For large commercial applications such as multimedia applications enhance processing power and most of the circumstances the overall application should be divided into different modules. These modules work in a close cooperation to complete the overall required task in less amount of time.

1.3.4 Parallel Computing Applications in Computer System

- In the recent developments in computing the techniques of parallel processing becomes easily adoptable because of growth in computer and commercial networks.
- The computations perform in parallel, by the use of collections of low power computing devices in the form of clusters. In fact of group of computers configured to solve a problem by means of parallel processing is termed as cluster computing systems.
- The advances in the cluster computing in terms of software frameworks made possible to utilize the computing power of multiple devices for solving of large computing tasks efficiently.

- The parallel computing is effectively applied in the field of computer security. The Internet based parallel computing set-up suitably used for the implementation of extremely large set of data required for computations.

1.4 Parallel Programming Platforms

- The basic components of the sequential computers are memory units and the processors. The logical representation of sequential computer is based upon the interconnection of memory and processors.
- In such systems, the memory unit is connected with the processor through the data path.
- This way we can understand about the overall processing of the computer system is based upon the basic components – memory, processor and data path.
- The processing capability of the system is improved because of the different architectural innovations tried in the computing field.
- The approach called as multiplicity is used for the performance improvement. The multiplicity is achieved by introducing multiple elements of memory unit, processing element and data path.
- There are variations in the use of multiplicity; one is in implicit parallelism where it is not seen by the programmer whereas it can be exposed to the programmer in different forms.
- This section focuses about useful architectural concepts applied in the field of parallel processing. The concepts and techniques covered here are applicable into varieties of platforms so that programmers can write different parallel computing programs.

1.4.1 Implicit Parallelism : Trends in Microprocessor and Architectures

- The field of computer architecture suggests designers about the selection and interconnection of hardware components for creation of computers that meet goals have been set for functionality, performance and cost.
- It is obvious thing that a new computer cannot be designed without defining the goals based on performance, power and costs.
- The overall design process is focused on understanding and making trade-offs. This include about the target market or users and what types of applications going to execute on these computers.
- This section describes about the understanding of various approaches of designs and their limitations as well as how it affects the overall performance of program developments.

1.4.1.1 Pipelining and Superscalar Execution

- The pipeline processor is analogically based on pipelining used in the car assembly line. The whole task to be performed in an assembly line is divided into sub-tasks. It was invented by Henry Ford long ago in the car assembly process.
- Henry Ford got an idea that it took much more time to build a car physically but he could actually build a car in a minute. What he did; all the steps required to build a car was divided into different stages on an assembly line. In this way one stage is assigned to put

engine, another stage is responsible to assign wheels, next stage is used to put cabinets and so on.

- When this logic was applied it still takes several hours to build first car but because overall assembly process is divided into different stages, the second car was just behind the first car it was almost done when the first car comes out from the assembly line.
- The 3rd car followed with the 2nd car and so on. In this way car assembly line was formed and mass production started.
- The basic logic used in the car assembly line is applied in the computer technology also but it is based on the workload instead of producing something physical.
- In computer pipeline the task to be performed is broken down into smaller subtasks being performed into different stages.
- We will understand about pipelining in computer by the use of an example. Suppose there is a requirement to add two numbers N1 and N2.
- When we think this from human brain points of view then we just look at the numbers and perform operation and write down the result. If these numbers are too big then we simply use calculator. The point here is we do not think about the process, we just apply the required operation.
- Now if we consider this thing in computer's points of view then while writing the program we are required to tell about the locations of these two numbers and what operation to be performed.
- In the similar fashion after performing the said operation we have to tell the computer where to store the result.
- The pipeline steps for performing addition operation on two numbers can be described with an example. The pipeline used for this example has four stages as Fetch N1, Fetch N2, Add N1 and N2, Store result.
- Now according to the modern computers, assume each of these operation use one clock cycle to complete. So collectively these take four clocks cycles for completion.
- However, the ability to perform operations in parallel while using the pipeline it improves the performance. This is further elaborated as while one instruction is being executed next instruction will be fetched and the overall output is actually improved. This can be illustrated better using pictorial representation as shown in Fig. 1.4.1

Unit

1 2 3 4

Clock cycles					
	1	Fetch N1	Inst-3	Inst-2	Inst-1
	2	Inst-4	Fetch N2	Inst-3	Inst-2
	3	Inst-5	Inst-4	ADD	Inst-3
	4				

Instruction 1 completed

Instruction 2 completed

Instruction 3 completed

Our Instrucon completed

Inst-6	Inst-5	Inst-4	Store Result

Fig. 1.4.1 Instruction Pipeline

- Here assume different instructions are represented using the name as Inst-#. Also one has to imagine that each Inst representing a stage involved in processing a computer instruction and each takes four clock cycles for completion.
- This example contains four units and in every clock cycles each unit does something. This can be further demonstrated by dividing the previous diagram shown in Fig .1.4.1 on the basis of units as shown in Fig 1.4.2.

- In every clock cycle each unit does something. This way we can easily understand that when instruction Inst-1 finishes by that time Inst-2 will become available to be finished in the next clock cycle.
- Each unit is processed in every clock cycle and in this way effectively utilization is achieved.
- Because all steps work together, four – step instructions (or tasks) can be completed at a rate of one per clock.
- Now the concept of superscalar execution can be covered after having sufficient understanding of pipeline processors.
- A processor designed to execute more than one instruction at a time during a single clock cycle is considered as a superscalar execution.
- A typical superscalar processor fetches and decodes several instructions at a time. The superscalar architecture exploits the potential of Instruction-Level Parallelism (ILP).
- The typical behavior of a superscalar processor is described with the fetching of multiple instructions at a time and then attempt for fetching of nearby independent instructions.
- These independent instructions can be executed parallel. The dependencies among instructions are identified so that instructions may be issued and executed in an order.
- The Fig 1.4.3 shows the general architecture of the superscalar processor.
- In Superscalar architecture multiple functional units are included. In this type of organization each unit is implemented with pipeline to provide support for parallel executions of multiple instructions.
- In the above examples five points functional units are considered where integer, floating – point and memory unit are included. These units are included. These units are used to

perform operations concurrently so that overall performance of the system is effectively improved.

Superscalar execution

- A superscalar machine is based on the Von-Neumann architecture but it can issue more than one instructions per clock cycle.
- A superscalar machine uses multiple pipelines because with a single pipeline it is not possible to issue multiple instructions.
- A superscalar processor can exploit any degree of instruction level parallelism. It depends upon the number of parallel blocks exists in the sequential code.
- This is made possible by increasing the spatial parallelism (building multiple execution units).
- The classifications of the superscalar processors are based upon the maximum number of instructions can be issued at the same time.
- This is depends on the pipeline structure used in the processor. One might clearly expect that the number of simultaneous instructions issued depends upon the number of pipelines built into the CPU.

1.4.1.2 Very Long Instruction Word Processors

- **VLIW** architectures are considered as one of the suitable alternatives to achieve Instruction Level Parallelism (ILP) in programs.
- This means VLIW architectures are used for execution of more than one basic instruction at a time in a program.
- VLIW architecture can store multiple instructions in a single word. In VLIW based systems a parallel compiler is used to generate operations to be executed in parallel in the same work.
- The compiler is responsible for resolving dependencies among instructions at compile time. The instructions ready to be executed concurrently are managed in a single word so that these can be executed on multiple functional units at the same time.
- Very large instruction word in VLIW architecture indicates that the program to be executed in such processors is to be recompiled in a way that the instruction runs sequentially without existence of stall in the pipeline.
- In this architecture it is not required for the hardware to examine the instruction stream about the instructions to be executed in parallel.
- The compiler determines which operations to be executed in parallel. The execution unit used by the operations is also determined by the complier.
- The performance of VLIW architecture is very good when sequential programs written in C or FORTRAN languages are executed after recompilation for such systems.
- The VLIW compiler ensures that all operations in executing unit can perform simultaneously.

- The VLIW processors are used in application area where high-performance is required with less cost. For example, Digital Signal Processing (DSP) applications are suitable for VLIW based architectures.
- VLIW architecture uses a fixed length machine language instruction format. This format is of fixed length but is much larger than 32 - or 64 – bit formats as shown in Fig 1.4.4. In VLIW every instruction is considered as a very long instruction. Each very long instruction contains bits to specify several instructions to be executed simultaneously. In a VLIW groups of bit fields are called as slots. These slots are used to specify operands and operations for every functional unit present in the machine.
- All of the slots present in the formats can be filled if the instruction level parallelisms present in the program are sufficient.
- The superscalar design and VLIW have almost similar effects. The different activity in VLIW is done in case of scheduling in which most or the entire scheduling task is decided statically at compile time by the compiler instead of runtime by the control unit.
- The data and resources dependencies associated in the program is handled and analyzed by the compiler.
- The compiler first analyzes the dependencies associated in the program and then packs the slots of each VLIW with as many concurrently executable operations as possible.
- In this way a VLIW architecture based machine can execute more instructions than a superscalar machine.

1.4.1.3 Basic Working Principle of VLIW Processor

The following are some of the points can be considered as principles on which VLIW processor works :

- VLIW processor's basic aim is to speeding up computation by exploiting instruction-level parallelism.
- VLIW uses the same hardware core as superscalar processors with multiple execution units(EUs) working in parallel.
- In VLIW processor an instruction consists of multiple operations in which are typical word length considered from 52 bits to 1 Kbits.
- In VLIW processor all operations in an instruction are executed in a lock-step mode.
- The VLIW processor relies on compiler to find parallelism and schedule dependency free program code.

1.4.1.4 Advantages of VLIW Processor

- Pure VLIW machines do not need complicated logic to check for dependencies.
- Eliminates complicated instruction scheduling and parallel dispatch associated with superscalar approach.
- Compiler is critical to performance : better compiler technology can result in improved performance on the same hardware.

1.4.1.5 Disadvantages of VLIW Processor

- Increased code size due to empty "slots".
- Increased memory bandwidth.
- Compiler is critical to performance : must do all dependency resolution.
- Cache misses in one pipeline will force all pipeline to stall in a "pure" VLIW machine.

1.5 Limitations of Memory System Performance

- The overall performance of a computer system not only depends upon the speed of the processing elements but significant memory unit working is also countable.
- This means processors has the capability for speed execution but how fast memory units feeds data to be processor is also considered.
- It is obvious that the speed of instructions executions by processors have been increased remarkably. The important fact here is that the execution speed highly depends upon the speed at which instructions and data supplied to the processor by memory components.
- It is really unfortunate that the access time of the memory components has not been improving as fast as the processor performance.
- The use of cache memory shows very useful impact on the overall performance of the system. In most of the modern architecture a memory system consists of different levels of cache.
- A cache is a high – speed memory to be used as the buffer memory. It is logically placed between the CPU and the main memory. The purpose of using cache memory is to hold instructions and data likely to be needed in the near future by the processor.
- It is required to decide what actually one has to measure apart from reading and writing associated with the memory systems.
- The first term needs to be highlighted here is the latency. It is the time elapses between the start of the operation and completion of the operation.
- There are enough reasons while saying that latency does not provide complete information about the performance of the memory system.
- At this particular point we need to discuss about the working of hardware. The understanding about hardware working makes clear why latency does not suffice as a measure of memory performance.
- The Fig. 1.5.1 shows a controller resides between the processor and the physical memory.
- The access of memory by the processor is handled by the controller as an intermediate entity. The processor sends its read or write request to the controller.
- The controller performs the translation of memory addresses and requests into appropriate signals for the underlying memory.
- Finally the controller passes these signals to the memory chips. The latency is minimized here when the controller returns an answer as fast as possible.
- This way we understand that memory latency is not fully sufficient to characterize as a measure for performance because it may need extra time between operations.

- The memory system performance can be assessed by measuring the speed at which a sequence of operations performed by the system.
- The term memory cycle time is used in this regard. The two parameters frequently used are read cycle time and the write cycle time in association with the memory system performance.

1.5.1 Use of Caches for Improvement of Latency

- The cache memory is used to enhance the access speed of any storage devices, for example, disk drives, main memory, tape storage, web servers and even for other cache also.
- The principle upon which cache works is called locality of reference. This principle says that the application refers a predictably small amount of data within a given window of time.
- The caches are divided into two basic groups : hardware and software cache which are shown in Fig. 1.5.2 and 1.5.3.
- The cache is the memory with low-latency and high-bandwidth properties. During functioning the required data by the processor is first fetched and stored in the cache memory so that further requests for the same data can be served from the cache.
- In this way the latency is reduced because the further fetch operations do not performed from the original locations.
- There are two terms frequently used here are cache-hit and cache-miss. When the requested data reference is satisfied by the cache is called cache-hit otherwise cache-miss.
- The computation rate of many applications depends upon the rate at which memory can provide data to the processor.
- Such applications' performances mainly depend on the cache hit and are called as memory bound.
- The cache memory comes with different capacity, actually the amount of data that can be stored in the cache is considered as the capacity of that cache, for example, 32KB cache.

The following are some of the cache related terms frequently used

- **Cache block** is the basic unit of cache storage. It can contain multiple bytes/words of data.
- **Cache set** is the term used to refer a row in the cache. The number of blocks per set is determined by the layout of the cache.
- **Tag** is uses to refer a group of data uniquely. An identifier is assigned for this purpose and is called as a tag. The tag is used to differentiate between different regions of memory mapped into the same block.

1.5.2 Memory Bandwidth

The rate of moving data between processor and memory is called as memory bandwidth. The memory bus and memory units are used to determine memory bandwidth. The memory bandwidth can be improved by increasing the size of the memory block.

1.5.3 Memory Latency Hiding Techniques

The increase in memory latency typically occurs when need arises to access remote memory. For example, a distributed shared memory based systems. The description of important latency hiding mechanisms used in the systems is mentioned in the following.

Using pre-fetching techniques

- The perfecting technique is used for latency hiding because it brings instruction or data close to the processor before their actual requirements. Perfecting techniques area classified based upon whether they are controlled by hardware or software.
- The perfecting techniques based on hardware control uses approaches like enhanced size of cache lines so that spatial locality of reference is reduced in multiprocessor applications.
- On other side explicit prefect instructions are required to issue when software controlled perfecting is used. While perfecting is done using software control the bandwidth requirements is reduced because perfecting is handled selectively.
- The direct effect of this scheme is that the time duration between the issue of instruction and its actual reference is increased. This has very significant impact when latencies are large.

1.5.3 Effects of Multithreading and Perfecting (Tradeoffs of Multithreading and Perfecting)

The problems associated with the performance of the memory system are minimized and in most of the cases eliminated by the use of multithreading and perfecting. The use of multithreading and perfecting affects the overall memory system performances.

1.6 Dichotomy of Parallel Computing Platforms

- Now we will have a brief description about the factors of parallel computing systems required for performance and portability in parallel programming.
- There are two different types of parallel platform organizations in a border sense. These are physical and logical organizations of platforms.
- When the platform is explored in terms of the parallel program design so that programmer focused on the program design only is referred as a logical organization. On

the other hand the physical organization deals with the hardware components and their interaction with each other to provide the platform for development.

- The programmer's point of view in parallel program development is based on the representation of parallel activities in terms of tasks and how various tasks communicate with each other to provide inter-task communication.

1.6.1 Control Structure of Parallel Platforms

- In general terms granularity refers about how a system is divisible. This means a system is considered as divided into multiple smaller parts is granular in structure. There are two basic types of granularity : course grained and fine grained.
- When a system is divided into large number of small parts we call it as fine grained whereas a course-gained is referred as the division of a system into smaller number of large parts. The parallel computing field uses the term granularity to describe about the division of a task into number of smaller subtasks.
- The fine grained granularity in parallel computing refers to the division of a task into a large number of subtasks usually of shorter duration whereas course grained is used to refer smaller number of subtasks.
- The granularity in a parallel program is considered at different levels for example program level and instruction level.
- A parallel program may be considered as a collection of programs or collection of instructions to be executed in parallel.
- This means number of parallel tasks in a parallel program execute independently in terms of various subtasks or various instructions of a program execute concurrently as separate tasks in parallel.
- The working of different processing units in a parallel computer is characterized by two approaches.
- First a single control unit is used to coordinate all processing units centrally and second approach is based on working of various processing units independently.

SIMD(Single Instruction Multiple Data Stream) Architecture.

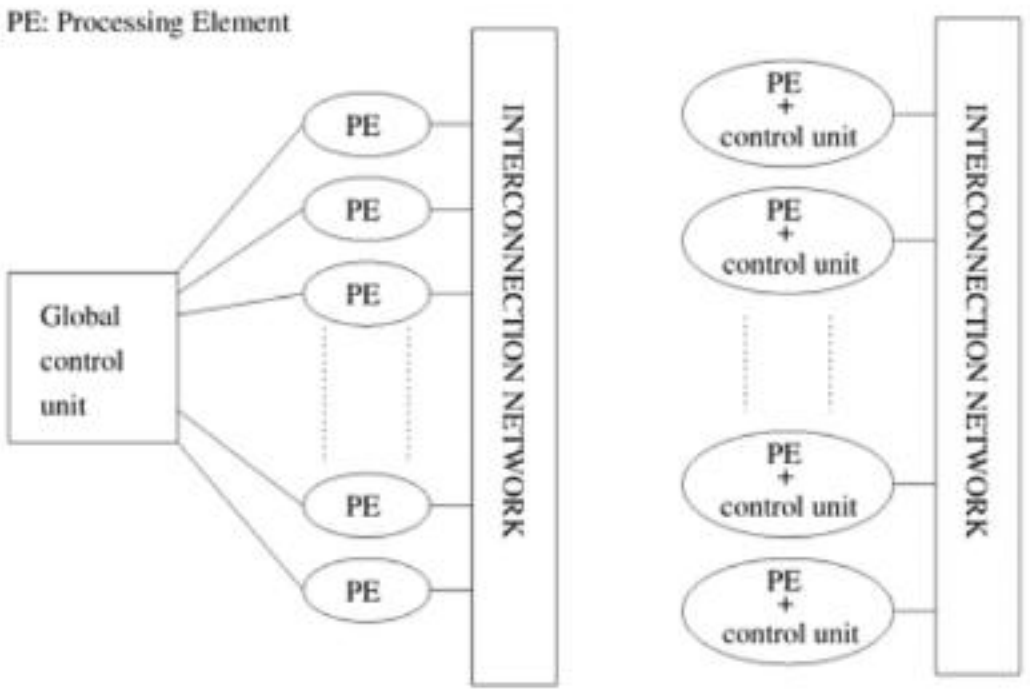
- In SIMD processors one instruction works on several data items simultaneously by using several processing elements (PEs), all of which carry out the same operation as shown in Fig. 1.6.1
- The SIMD model of systems uses a single control unit to dispatch multiple instructions to various processing elements.
- In the SIMD computing model a single control unit is used to read instructions by a single Program Counter(PC), decode them and send control signals to the PEs. The number of data paths are depends upon the number of processing elements.

- Data are to be supplied to and derived from PEs by the memory. The structure of the system is shown in Fig. 1.6.2 called as array processor.
- The difference units like PEs and memory modules are interconnected networks. Example : Execution of conditional statements on a SIMD Architecture.
- The following program segment where a conditional statement is included is considered to illustrate the use of SIMD architecture.

```
if(y == 0){

        z = x;

}

else{

        z = x/y;

}
```

- This statement is executed in two steps. In the first step all the processors where the instruction. z = x is present execute this instruction whereas all other processors remain idle. In the second step else part of the segment will execute and the processors involved in the first step remain idle. Fig. 1.6.4 shows the execution of the conditional statement two steps..

- **Figure 1.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).**



(a) (b)

Message-Passing Platforms

The logical machine view of a message-passing platform consists of p processing nodes, each with its own exclusive address space. Each of these processing nodes can either be single processors or a shared-address-space multiprocessor - a trend that is fast gaining momentum in modern message-passing parallel computers. Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.

On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name message passing. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes.

In its most general form, message-passing paradigms support execution of a different program on each of the p nodes.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are `send` and `receive` (the corresponding calls may differ across APIs but the semantics are largely identical).

In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification or ID to each of the multiple processes executing a parallel program. This ID is typically made available to the program using a function such as `whoami`, which returns to a calling process its ID.

There is one other function that is typically needed to complete the basic set of message-passing operations - `numprocs`, which specifies the number of processes participating in the ensemble. With these four basic operations, it is possible to write any message-passing program.

Different message-passing APIs, such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), support these basic operations and a variety of higher level functionality under different function names. Examples of parallel platforms that support the message-passing paradigm include the IBM SP, SGI Origin 2000, and workstation clusters.

It is easy to emulate a message-passing architecture containing p nodes on a shared-address-space computer with an identical number of nodes. Assuming uniprocessor nodes, this can be done by partitioning the shared-address-space into p disjoint parts and assigning one such partition exclusively to each processor.

A processor can then "send" or "receive" messages by writing to or reading from another processor's partition while using appropriate synchronization primitives to inform its communication partner when it has finished reading or writing the data.

However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another node's memory requires sending and receiving messages.

1.7 Physical Organization of Parallel Platforms

In this section, we discuss the physical architecture of parallel machines. We start with an ideal architecture, outline practical difficulties associated with realizing this model, and discuss some conventional architectures.

1.7.1 Architecture of an Ideal Parallel Computer

A natural extension of the serial model of computation (the Random Access Machine, or RAM) consists of p processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. Processors share a common clock but may execute different instructions in each cycle. This ideal model is also referred to as a parallel random access machine (PRAM). Since PRAMs allow concurrent access to various memory locations, depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

1. Exclusive-read, exclusive-write (EREW) PRAM. In this class, access to a memory location is exclusive. No concurrent read or write operations are allowed. This is the weakest PRAM model, affording minimum concurrency in memory access.
2. Concurrent-read, exclusive-write (CREW) PRAM. In this class, multiple read accesses to a memory location are allowed. However, multiple write accesses to a memory location are serialized.
3. Exclusive-read, concurrent-write (ERCW) PRAM. Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized.
4. Concurrent-read, concurrent-write (CRCW) PRAM. This class allows multiple read and write accesses to a common memory location. This is the most powerful PRAM model.

Allowing concurrent read access does not create any semantic discrepancies in the program. However, concurrent write access to a memory location requires arbitration. Several protocols are used to resolve concurrent writes. The most frequently used protocols are as follows:

- Common, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- Arbitrary, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- Priority, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- Sum, in which the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

1.8 Communication Costs – Overhead for communicang between processing elements.

1 Message Passing Costs in Parallel Computers

- Communication me = Message prep me + Network traversal me
  - **Startup me** – me required to handle the message at the sending and receiving nodes.
  - **Per-hop me** (or **node latency**) – time taken by the header of a message to travel between two directly-connected nodes.
  - **Per-word transfer me** – 1 / *r* where *r* is channel bandwidth (*r* words per second)
- Types of roung
  - **Store-and-forward roung** – Each intermediate node on the path forwards the message to the next node only aer it has received and stored the entire message.
    - Linear me – $t_{comm} = t_s + (mt_w + t_h) * L$
    - Parallel me – $t_{comm} = t_s + mLt_w$
  - **Packet roung** – Break up the message into smaller packets and then forward them through the network
    - Time – $t_{comm} = t_s + t_hL + t_wm$ where $t_w = t_{w1} + t_{w2}(1 + s/r)$
  - **Cut through roung** – Assign messages predefined routes based on message type then transfer as "its"
    - Time – $t_{comm} = t_s + Lt_h + t_wm$
- Cut-through roung can be opmized using the following techniques
  - Communicate in bulk – save on startup me by sending larger messages
  - Minimize the volume of data – reduce per word overhead
  - Minimize distance of data transfer – minimize number of hops

2 Communication Costs in Shared-Address-Space

- Associang communicaon costs with parallel programs is harder in message passing architectures than in shared-access-space architectures. This is because in message passing systems:
    - Memory layout is determined by the system
    - Finite cache sizes can result in cache thrashing
    - Overheads associated with invalidate and update operaons are dicult to quanfy
    - Spaal locality is dicult to model
    - Perfecng can play a role in reducing the overhead associated with data access.
    - False sharing is oen an important overhead in many programs
    - Contenon in shared accesses is oen a major contribung overhead in shared address space machines.

1.9 Scalability Design Principles

1. Avoid the single point of failure. We can never just have **one** of anything, we should always assume and design for having at least two of everything. This adds costs in terms of addional operaonal eort and complexity, but we gain tremendously in terms of availability and performance under load. Also, it forces us into a distributed-rst mindset. If you can't split it, you can't scale it has been said by various people, and it's very true.
2. Scale horizontally, not vercally. There is a limit to how large a single server can be, both for physical and virtual machines. There are limits to how well a system can scale horizontally, too. That limit, though, is increasingly being pushed further ahead. Even databases are moving in that direcon. Furthermore, the cost of (vercally) *upgrading* a server increases exponenally whereas the cost of (horizontally) *adding* yet another (commodity) server increases linearly.
3. Push work as far away from the core as possible. There are several orders of magnitude more clients than servers as we move inward into the core of our applicaon. The less work the few have to do on behalf of the many, the beer. The smaller the updates we can pass to our clients, the beer.
4. API rst. In addion to pushing work to the clients, view your applicaon as a service with an API rst. Clients these days are smartphone apps, web sites with JavaScript, and desktop applicaons. If the API does not make assumpons about which clients will connect to it, it will be able to serve all of them. And you open your service up for automaon, as well.
5. Cache everything, always. Caches are essenally storages of precomputed results that we use to avoid compung the results over and over again. This is a Godsend for scalability and performance, so we must use it.
6. Provide *as fresh as needed* data. Depending on your applicaon, users might not need the very freshest data right away. Eventual consistency leads to much better availability under the CAP theorem. If we actually need strict consistency, so be it.
7. Design for maintenance and automaon. Soware needs monitoring and updates to ensure proper operaon over time. As we move out of the servers are pets era and into the servers are cale era, our mindset has to change. Do we even need to recongure servers anymore? Can't we just take old ones down and replace with new ones, that have been congured oine as part of their image creation? What monitoring data is important to us? What informaon does it provide us with? Do not under-esmate the me and eort spent in maintaining your

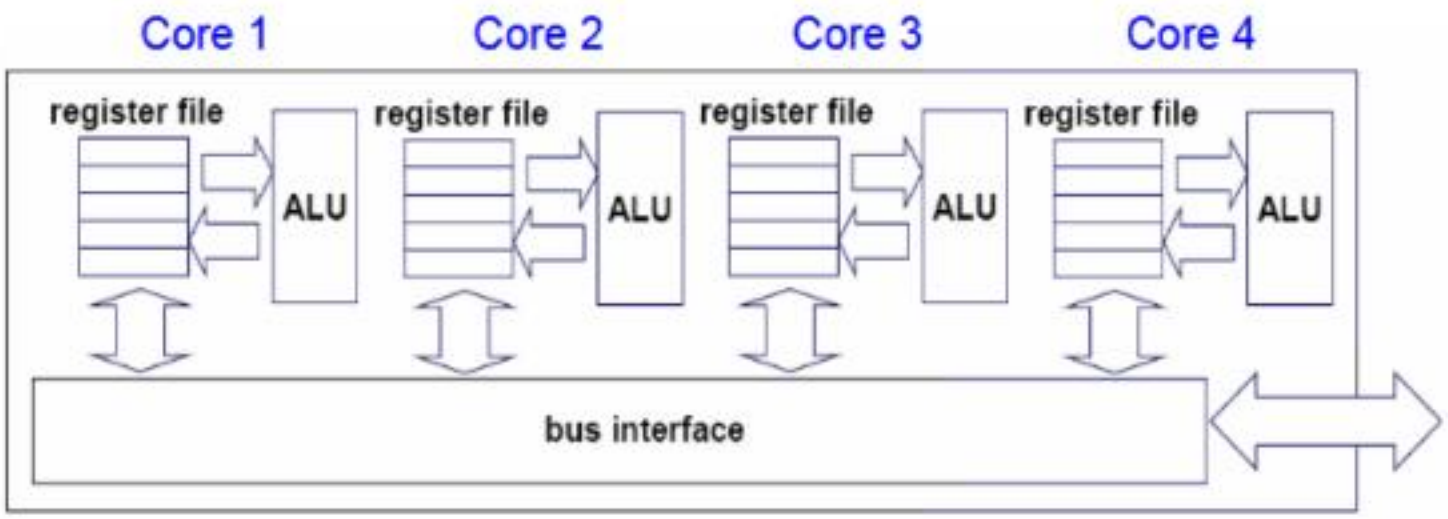
applicaon. Your inial public soware release is a laudable milestone, it also marks when the **real** work begins.

8. Asynchronous rather than synchronous. We already understand asynchronous communicaon perfectly in the physical world. We drop o a leer in the mail, and some me later, it arrives. Unl it does, we convince ourselves that it is underway, oblivious to the complexity of the postal system. We only use personal couriers for very important messages. A similar approach should be taken for our applicaons. Did a user just hit submit? Tell the user that the submission went well, and then process it in the background. Perhaps show the update as if it is already completely done in the mean time.
9. Strive for statelessness. While it may seem tempng to avoid inter-component communicaon by keeping track of certain state informaon in e.g. your applicaon servers, don't. Unless we host purely stac pages, we can never get away from state informaon. We must make sure that state informaon is kept in as few places as possible, and within components made for it. Web and applicaon servers are not, but distributed key-value stores are. Keeping it there lets you treat your web and applicaon servers as completely replaceable instances, which is ideal from a scalability point of view since your server eet can much more easily be modied when any server is able to handle any dient request (despite the dient being in the midst of a "session").
10. This too shall fail. Computer systems fail. Soware fails. Hardware fails. Designs fail. Failure handling fails! Be prepared for failure, but spare end users from witnessing it too obviously. It reects poorly on you, even if failure is inevitable.

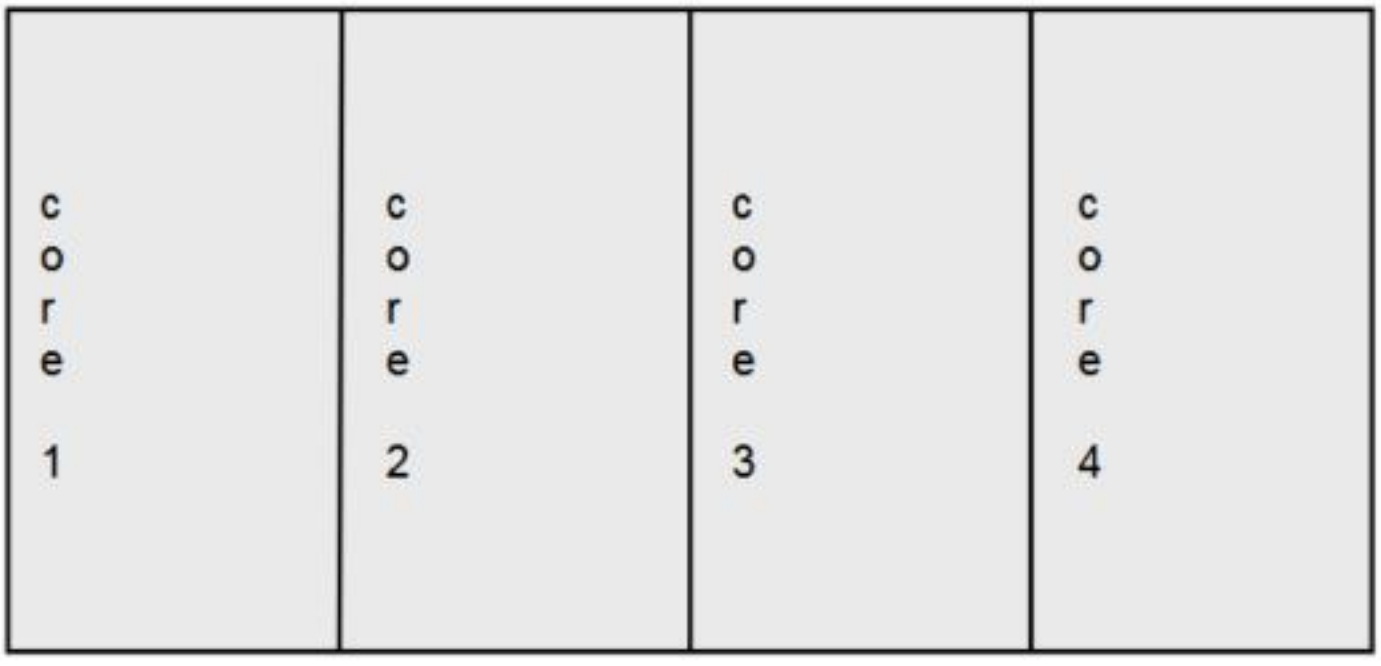
1.9 Multi-core architectures

- Deeply pipelined circuits:
  heat problems
  speed of light problems
  difficult design and verification
  large design teams necessary

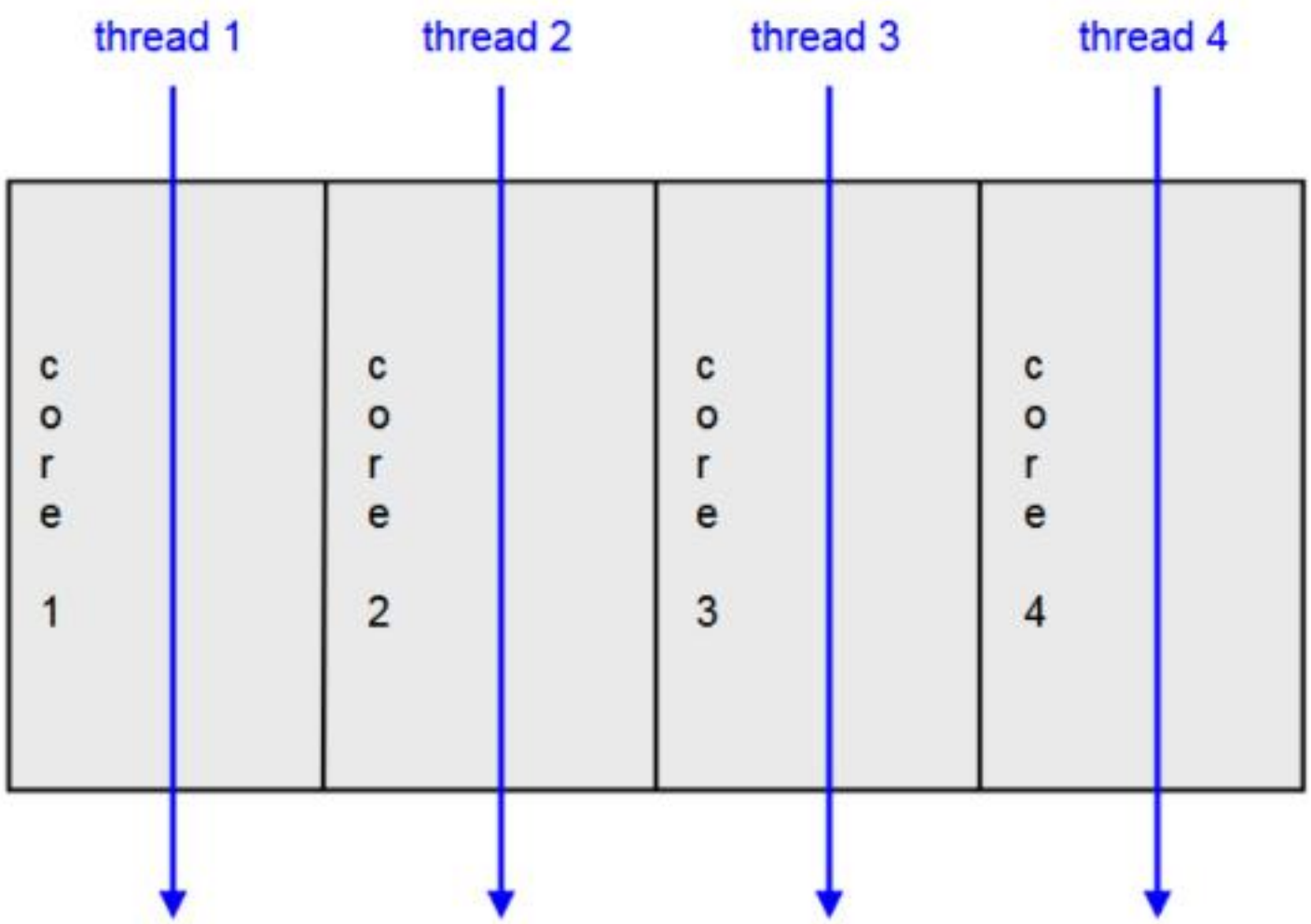
- Many new applications are multithreaded
- General trend in computer architecture



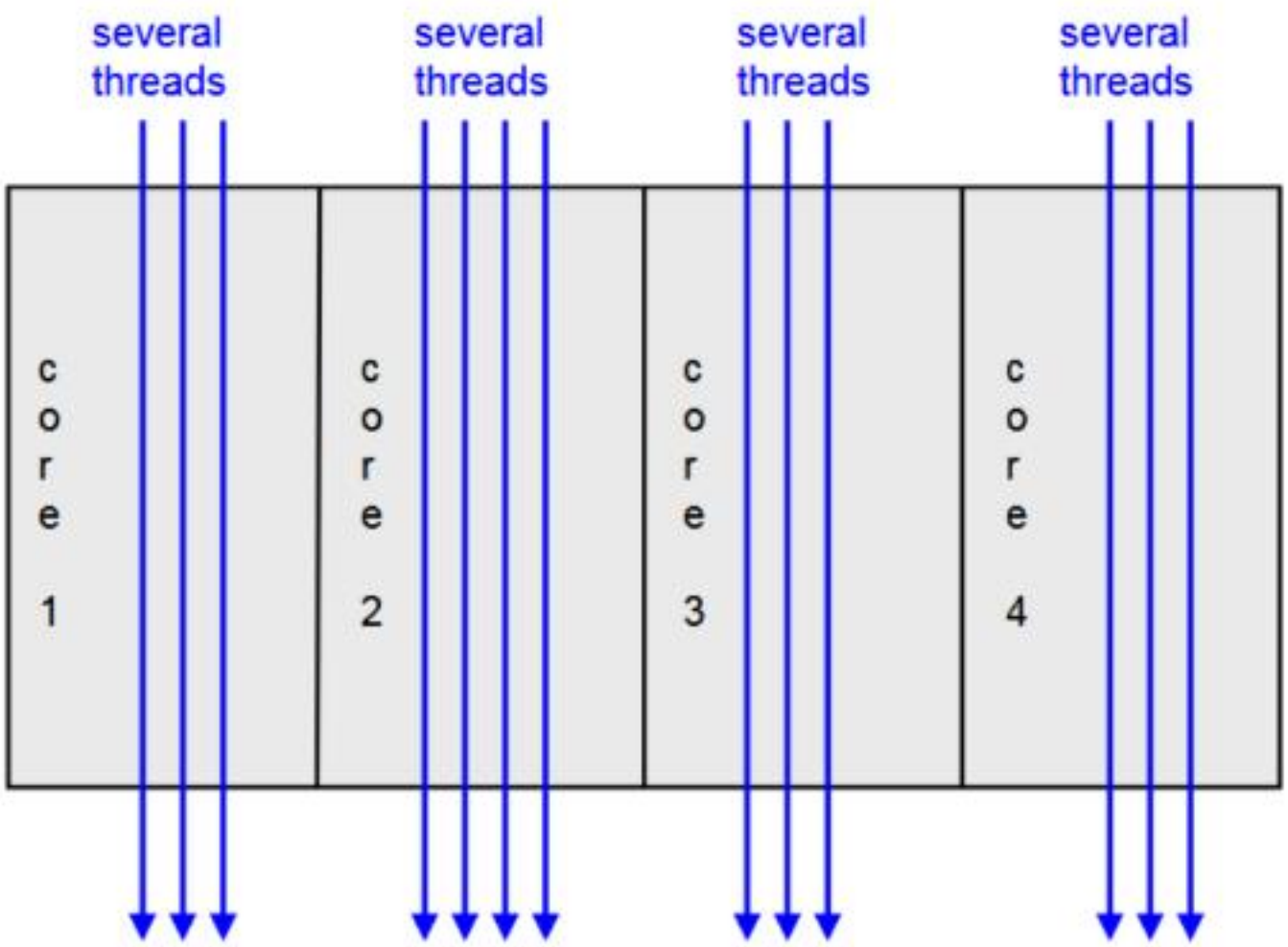
- Replicate multiple processor cores on a single die.
- The cores fit on a single processor socket.



The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with OS

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today

UNIT – II Parallel Programming

2.1 Principles of Parallel Algorithm Design

An algorithm provides step by step solution of the given problem. An algorithm accepts inputs from the user and based upon the input after performing the defined computations provides the output. In the similar fashion particularly a parallel algorithm accepts inputs from the user and execute several instructions simultaneously on different processing units and all separate outputs produced from different units are combined to provide the overall output of the algorithm. In this chapter we will discuss several techniques required to deal with the design of parallel algorithms.

2.1.1 Preliminaries

- The basic steps in the design of parallel algorithms are : partitioning of overall computation into smaller computations and assignments of these smaller computations into different processors.
- This two key terminologies are explored in this section and some of the basic terminologies associated with the parallel algorithm design are also focused at significant levels.

2.1.2 Decomposition, Tasks, and Dependency Graphs

- Decomposition : The computer system is used to solve a problem by performing some computations based on input data and provide output data for further activities. The overall computation can be partitioned into number of small size computations so that these computations execute in parallel. The decomposition deals with the approaches of partitioning the overall computations into sub parts. When a computation is divided into many small tasks, it is referred as fine – grained decomposition. On the other hand the course – grained decomposition contains a small number of large tasks.
- Tasks : As we know that the Operating System provides an environment for program developments and executions. In fact it is considered as an overall controller of the program. In programming the basic unit of computation is referred as a task and is controlled by an operating system. In the context of parallel programming the tasks are units of computation based upon that the overall computations is decomposed. The problem to be solved using the parallel programming approach is divided into arbitrary sized multiple tasks for simultaneous executions so that the computation time is minimized effectively.
- Task-dependency graph : Is a directed acyclic graph. Typically a graph is a collection of nodes and edges, the task – dependency graph also contains nodes and edges. The nodes in the task – dependency graph represent tasks whereas edges between any two nodes represent dependency between them. For example, there is an edge exists between two nodes T1 and T2, if T2 must be executed after T1.

Ex2.1.1 : Consider the task – dependency graph shown in Fig 2.1.1 and find – out the following :

i. Maximum degree of concurrency
ii. Critical path design
iii. Total amount of work
iv. Average degree of concurrently

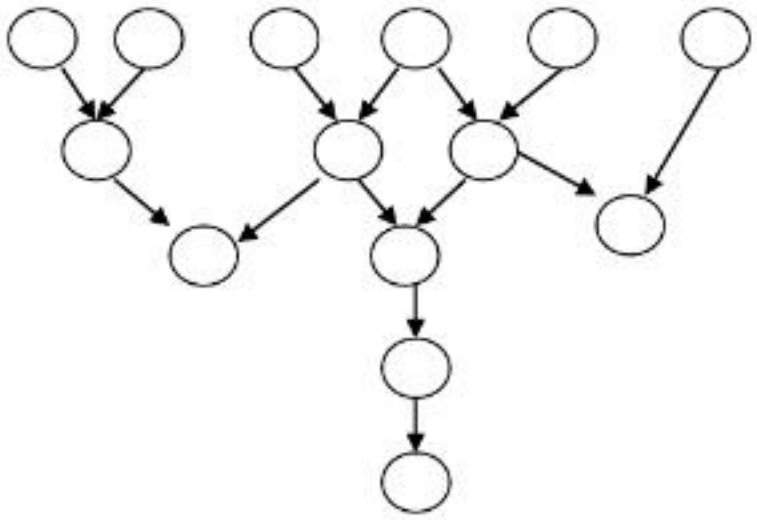


Fig. 2.1.1 Task-dependency graph

Solution :

i. **Maximum degree of concurrency** : The maximum number of tasks allowed to execute in parallel is termed as the maximum degree of concurrency. In the given graph the maximum degree of concurrency is 6.

ii. Critical path design :

- There are two types of nodes included in the task – dependency graph namely start node and finish node. Start nodes are nodes with no incoming edges whereas nodes with no outgoing edges are called as finish nodes.
- The critical path in a task – dependency graph is the longest directed path between any pair of start and finish nodes. The quality referred as critical path length is the sum of the weight of the nodes on a critical path.
- In the given graph the critical path length is 5.

iii. **Total amount of work** : Here, the total amount of work is 14 if it is assumed that the each task takes one unit of time.

iv. **Average degree of concurrency :** The average number of tasks allowed to execute in parallel is called average degree of concurrency. It is a more useful measure and is computed using the following formula

(Average degree of concurrency → Total amount of work/Critical path length)

In this example average degree of concurrency – 14/5 → 2.8

Example : Data Decomposition

- The matrices A and B are divided into number of rows and columns. The divisions of rows and columns are in continuous sequence and termed as strips. The matrices A and B are multiplied and the resultant matrix C is generated. The elements of the C matrix can be computed independently according to the definition of matrix multiplication.
- This fact suggests the possibilities for performing matrix multiplication in parallel. The parallel computations can be performed using dividing the overall computation into number of subtasks.
-
- In this case each subtask should contain a row of the matrix A and a column of the matrix B. The total number of subtasks included to perform the multiplication appears to be equal to N^2 according to the elements in resultant matrix C.
- The necessary computations of the basic subtasks can be performed when the required data sets are available. The required data sets are a row of the matrix a and all the columns of matrix B for basic subtasks.
- The simple solution in this case is duplicating the matrix B in all the considered subtasks for multiplication. This solution is unacceptable because of the extra memory requirements for extra data storage. The solution for this problem should have the data availability only required for the computations.
- The algorithm for the matrix multiplication with the solution of the problem mentioned here is an iterative procedure. In this procedure the number of iterations is equal to the number of subtasks. In this case at each iteration of the algorithm a row of matrix A and a column of matrix B are contained in each subtask.
- The subtasks containing rows, and columns computes scalar products at each iterations and the corresponding elements of the matrix C are generated. After completing of all iteration computations the columns of matrix B must be transmitted so that subtasks should have new columns of the matrix B and new elements of the matrix C could be calculated.
- This transmission of columns among the subtasks must be executed in such a way that all the columns of matrix B should have appeared in each subtask sequentially.
- Fig 2.1.1 shows the iterations of the matrix multiplication algorithm where four rows and four columns are assumed. In this scenario, at the beginning of the computations each subtask contains i-th row of A matrix and i-th column of B

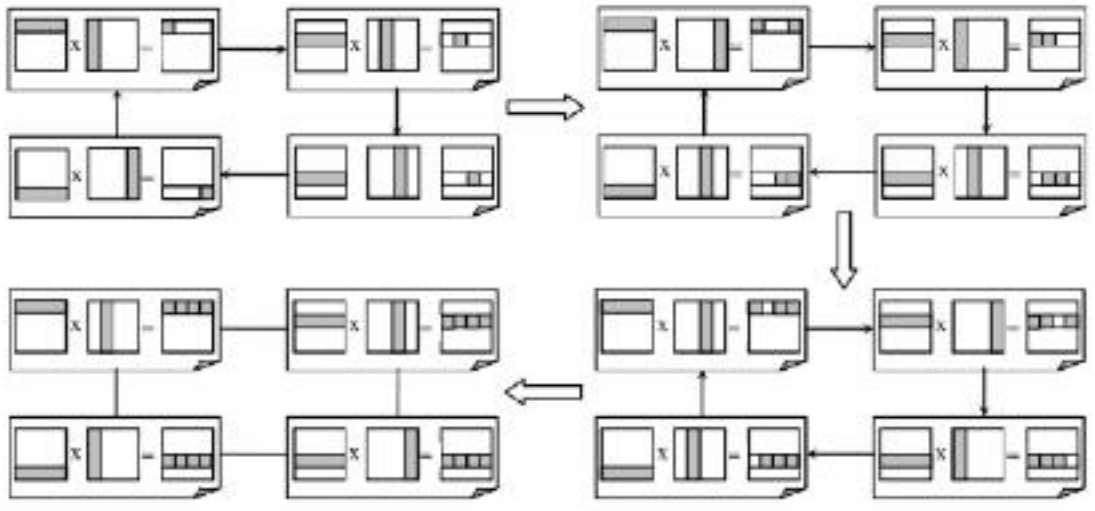


Fig 2.1.1 General scheme of data communications for the parallel matrix multiplication algorithm using block striped decomposition

- As a result the subset i can compute the element C_{ij} of the result matrix C. Further each subtask transmits its column of matrix B to the following subtask in accordance with the ring structure. These actions should be repeated until the iteration of the parallel algorithm are completed.

2.1.3 Granularity, Concurrency and Task-Interaction

Granularity

- Granularity is the descriptions about a system in terms of how divisible it is. There are two types of granularity namely fine-grained and course-gained. Fine-grained system has a high-granularity. This means in fine-gained an object(or system) is divided into larger numbers of smaller parts.
- In the similar way a course-grained refers to the system divided into a smaller number of larger parts. For example, the use of grams to represent weight of an object is more granular than using kilograms to represent the weight of the same object.
- In general parallelism is categorized with instruction and data stream in parallel computing. Granularity in parallel computing is an additional way of categorizing parallel computations. Term granularity is used in parallel computing is an additional way of categorizing parallel computation. Term granularity is used in parallel computing to refer about the division of overall task into number of subtasks.
- The fine-gained in parallel computing refers the division of a task into a large number of smaller subtasks. In the similar fashion course-grained describes the division of a task into the larger number of longer subtasks. In parallel computing, technically a measure of the computational work is called as a grain. In fact a grain represents the ratio of computation work to communication work.

- How to measure granularity : There are three relative values actually used to specify the granularity : fine, medium or coarse. In the parallel algorithm the granularity is generally used to describe about a parallel section of algorithm.
- There are three characteristics of the algorithm used for the running of the algorithm. The first characteristics is the structure of the problem. The data parallel programming works in the form where specified operations are performed on many pieces of data.
- These operations are executed by different processing elements and communications amongst the PEs required. In association with granularity the task in data parallel computation is considered to have small granularity or fine-grained. On other cases like the execution of large subroutines independently with little communication with each other is considered as coarse-grained.
- The second characteristic is the size of the problem. This can be described using the example where 10 numbers are to be incremented. The number of processing elements in this case is assumed as 10 then the algorithm requires 1 clock cycle for computation.
- Later on assume the increase in the problem size and therefore 100 numbers are to be incremented. This way the task size has been increased and this is considered as a coarser granularity.
- The third characteristic to deal with the granularity in the parallel algorithm is the number of processors available. The number of processors is reduced for the problem size the task size increases and the granularity becomes coarser.

Concurrency

- The tendency for the events in real world to happen at the same time is called as concurrency. The concurrency is one of the natural phenomena in the real world because at a particular instance of time many things are happening simultaneously. The concurrency is required to deal with for designing the software system for real world problems.
- There are generally two important aspects when dealing with concurrency for real world problems : ability of dealing and responding external events occur in random order, and required to respond these events in some minimum required interval.
- It is really simple to handle when different activities happen in a truly parallel fashion by creating separate programs to deal with each activity. The real challenge occurs when required to design concurrent system in which interactions among the concurrent activities need to be coordinated.
- The concepts termed as degree of concurrency is related to granularity is used in the algorithm design. The maximum degree of concurrency is the maximum number of concurrent tasks can be executed simultaneously in a program at any given instance of time.
- Another term related to concurrency is the average degree of concurrency which is used to refer the average number of tasks can be processed in parallel during the execution of the program.

Task-Interaction

- The parallel executions of tasks provide efficiency in terms of speedup because tasks in a parallel algorithm simultaneously execute in different processors. There are basically three factors granularity, concurrency and interaction are responsible to affects the speedup of the parallelization.
- The interaction provides the communication among the tasks running in different processors. The interactions among the tasks included for providing the solution of a problem is required because the tasks share input, output, or intermediate data.
- The task-dependency graph shows the dependencies because output of one task can be the input of another task. This means the second task cannot proceed unless the output of the first task becomes available.

2.1.4 Processes and Mapping

- A problem to be solved using the parallel programming approach is considered to be divided into number of parts. Each part can be performed independently and is called as a task. In the context of this discussion a process is an entity responsible for performing the assigned task. This process is an abstract idea about doing the computations based upon the steps included in a task and input data available for it producing the output.
- The environment in which a problem is being solved by means of parallel computations based usually has several running process simultaneously. These processes communicate with each other for the need of synchronization and exchanging of information.
- The program written for such platforms are called as parallel programs and one parallel program can have several processes for handling different tasks simultaneously.
- The Fig. 2.1.2 describes a problem divided into several tasks and assigned to separate processors for simultaneous executions.

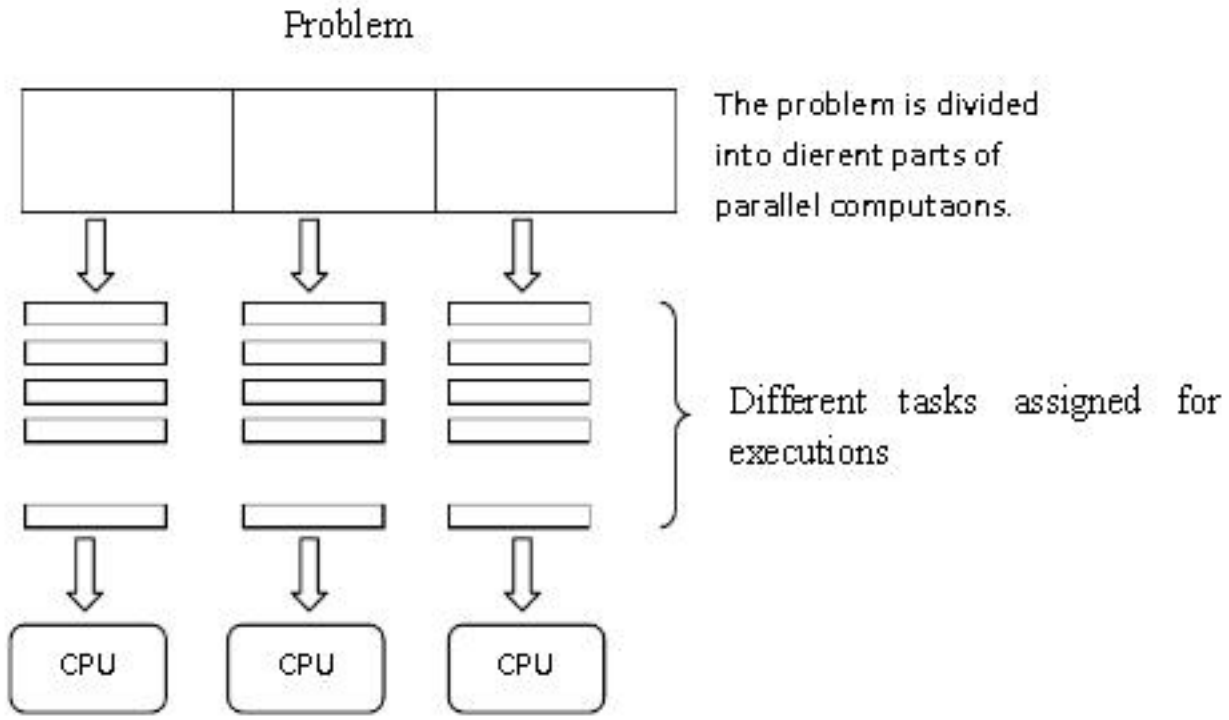


Fig. 2.1.2 Parallel execution of tasks

2.2. Decomposition Techniques

- The problem required much more time for computations are possible to divide multiple parts and each and every part is the implementation of a particular task.
- The task-dependency graph used to design a set of tasks and these tasks are used the candidate for concurrent executions. There are various techniques commonly used for decomposition and in this section some of the techniques are covered.

2.2.1 Data-Decomposition

- The large data sets involved in the problems can be divided into smaller parts and processed independently by several computers concurrently. The computations of partitioned data on computers are usually used for some kinds of analysis.
- The problem is classified as an embarrassingly parallel if the subset of the data can be analyzed independent of the rest of the data. The data-decomposition is a common technique used for concurrent processing of the data.
- There are basically two steps used in this technique. In the first step the overall input data required for performing the defined computation is divided into multiple parts. The second step is based upon the division of overall computation into number of tasks handled on the portioned data.
- The actual operations implemented on these tasks are similar but the data upon which these operations work are different.

Example : Matrix Multiplication

Consider the problem of multiplying two n x n matrices A and B to yield matrix C. The output matrix C can be partitioned into four tasks as given below. Here each task computes one element of result matrix.

$$\text{Matrix A} \rightarrow \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} \qquad \text{Matrix B} \rightarrow \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$$

$$\text{Matrix C} \rightarrow \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix} X \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$$

$$\text{Matrix C} \rightarrow \begin{pmatrix} c11 & c12 \\ c21 & c22 \end{pmatrix}$$

$$\begin{pmatrix} & \\ & \end{pmatrix}$$

a11.b11+a12.b12 a11.a12+a12.b22

a21.b11+a22.b21 a21.b12+a22.b22

2.2.2 Recursive Decomposition

- As we know that the Divide and conquer is one of the strategy of solving the computational problems. The divide and conquer strategy is based on two ideas : first approach says to solve the problem directly if it is trivial and in the second approach, decompose the problem into smaller parts if it cannot be solved as it is and solve the smaller parts.
- The recursive decomposition is based on providing concurrency in problems that can be handled in divide and conquer strategy.
- The problem to be solved using recursion is divided into multiple sub problems provided that each of these sub problems is independent. Each sub problem can be solved individually by dividing further into other sub problems and solved using recursion. In programming a function on procedure calls itself is called as recursion. For example in the following program the function demo is a recursive procedure.

```
main()

{

        int i;

        i = 10;

        demo(i);

}

demo(int count)

{

        count--;

        prin("The value of the count is %d\n", count);

        if(count > 0)

        demo(count);

        prin("The value of the count is %d\n", count);

} •
```

Example : Merge sort

- The example of merge sort as shown below is implemented in a sequential and parallel version thereafter. In this example the array is first divided into two parts then sorted these two parts recursively. The sorted parts are finally merges to produce the final output. The overall computations are organized here in the binary trees.
- The parent process provides array to each process for sorting of elements. The process divides the array into two halves and sends to the children. The children perform their part of the computations and send the sorted array back to the parent. The merging of these elements is performed by the parent and sends the array back up in the tree.

Merge sort : Pseudo code for sorting of an array using merge sort sequentially.

```
void mergeSort(int *a, int rst, int last, int *aux)

{

        if(last <= rst)

        return;

        int mid = (rst/last)/2;

        mergeSort(a, rst,mid,aux);

        mergeSort(a,mid+1, last, aux);

        mergeArrays(a,rst,mid,a, mid+1,last, aux,rst,last);

        for(int i=rst;i<=last;i++)

        a[i]=aux[i];

}

void mergeArrays(int *a, int arst, int alast, int *b, int brst, int blast, int *c, int crst,int clast)

{

        int i = arst, j=brst, k=crst;

        while(i<=alast && j<= blast)

        {

                if(a[i]<b[j])

                c[k++] = a[i++];

                else

                c[k++] = b[j++];

        }
```

```
while(i<=alast)
        c[k++] = a[i++];
        while(j<=blast)
        c[k++] = b[j++];
}
```

MergeSort : Pseudo code for sorting of an array using mere sort in parallel

```
void parallel_mergeSort()
{
        if(proc_id > 0)
        {
                recv(size,parent);
                recv(a,size,parent);
        }
mid = size/2;
if(both children)
{
                send(mid, child1);
                send(size-mid,child1);
                send(a,mid,child1);
                send(a+mid, size-mid,shild2);

                recv(a,mid,child1);
                recv(a+mid, size-mid,child2);
                merge Arrays(a,0,mid,a,mid+1,size,aux,0,size);
                //declare aux local
                for(int i=rst; i<=last; i++)
                a[i] = aux[i];
                }
```

```
else
                    mergeSort(a,0,size);
                    if(proc_id > 0)
                    send(a,size,parent);
}
```

2.2.3 Exploratory Decomposition

There are situations where the problem decomposition goes hand in hand with its executions. Such problems typically involve the exploration or search of a state space of solutions. The search space of the problem is divided into smaller parts and each smaller part is searched concurrently till the point at which the expected solution is found.

Example :

- As an example of exploratory decomposition consider a stage space searching problem such as finding a solution to a puzzle problem. The stops of solving such problems using the exploratory decomposition are as follows:
- The computations required for the decomposition can be divided into multiple tasks where each task is searching for a different portion of the search space.
- The task of finding the shortest path from initial to final configuration now translate to finding a path from one of these newly generated nodes to final configuration.
- The 15 puzzle problem is typically solved using tree search techniques. Starting from initial configuration, all possible successors are generated.
- In the 15 puzzle problem there are 15 tiles numbered 1 through 15 are arranged in a 4 x 4 grid as shown in Fig 2.2.1. One tile is left blank so that moves can be made. The four possible moves here are represented as moves up, down, left, right. The initial as well as the final configuration was specified.
- This problem is characterized by the objective of determining any sequence of moves or a shorter sequence of moves. The solution of the problem must be searched from an arbitrary state.

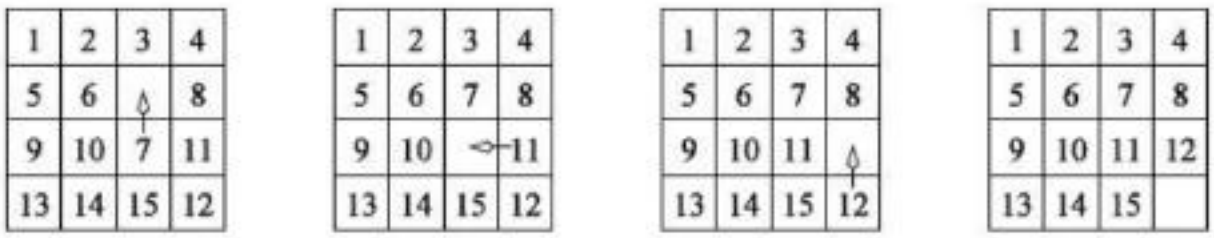


Fig. 2.2.1

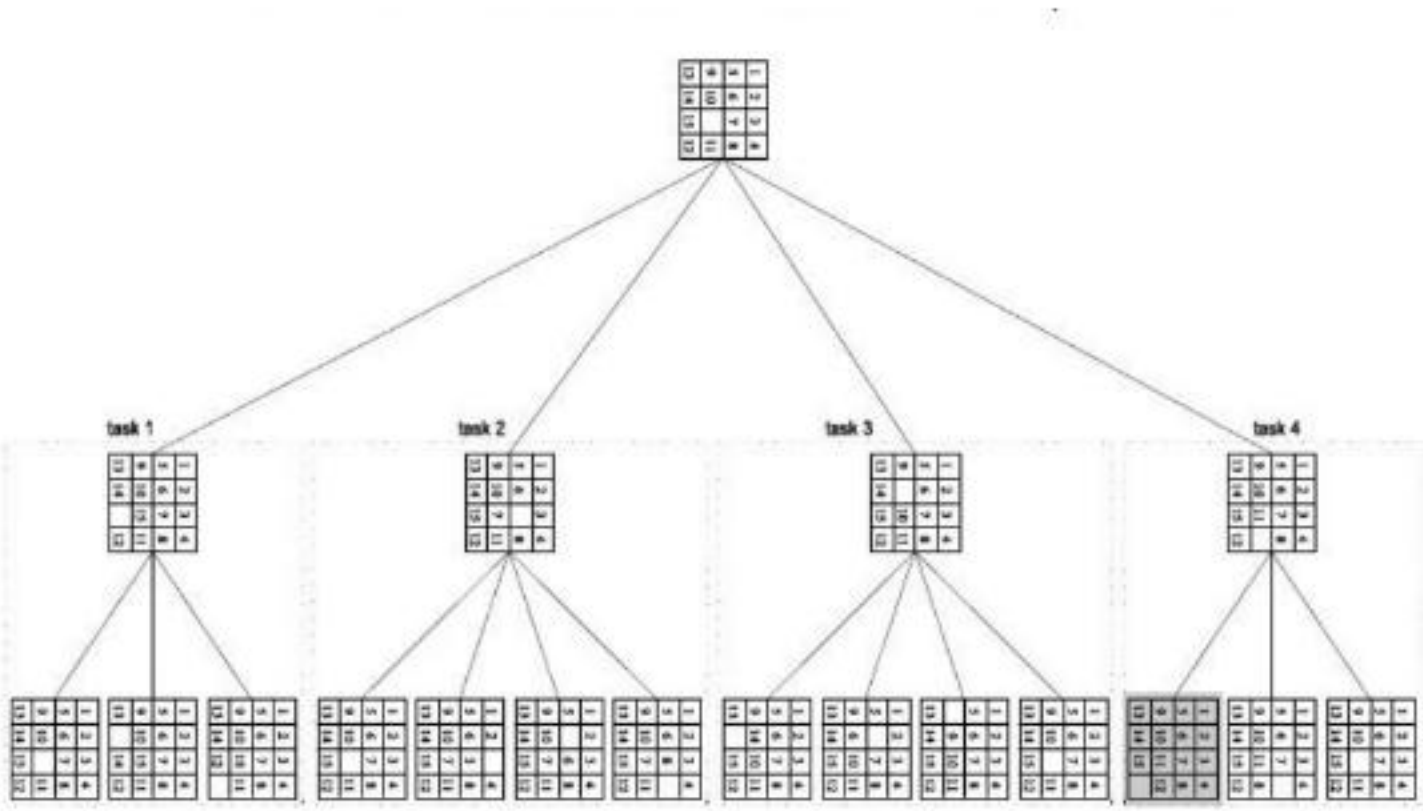


Fig. 2.2.2

- The solution of the 15 puzzle problem is provided using the tree – search techniques. Once we start from the initial configuration all the configurations possible as successor configuration are generated. This is handled using two possibilities : one says about to occupy empty slot by any one of the neighbor present, and second is to find out a path from one of the new configuration to the final one.
- A state space graph : Fig 2.2.2 shows the configuration space generated by the tree search As we know that any graph is a collection of nodes and edges the state space graph also contains nodes and edges. In this graph the nodes are used to represent configurations and edges represent connections of configurations. Here every edge of the graph connects configuration that can be reached from one another by a single move of a tie.

2.2.4 Speculative Decomposition

- The speculative decomposition technique is used in a situation when a program has many options to take in terms of branches based upon the outputs of other parts that preceded it. The situation can be described in terms of specific task, consider a task T1 involves in doing the computation C1 and going to produce the output O1.
- The next computation to be performed may be decided based upon the output of task T1. In the similar fashion other tasks can start executions concurrently in the next stage. The situations can be better compared with the switch statement in C programming.

- The switch statement works based upon the value of the expression on which it is based and only the corresponding case statement executes. Some or all of the cases execute in advance in speculative decomposition.
- The situation in which value of the expression is known then the results from the computation to be executed in that case will be kept. The anticipation about the possible computations causes the performance gain in the statement executions. For example consider the following switch statement in sequential and parallel versions.

Sequential	Parallel version

compute expr;
switch(expr)
{
case 1 : compute-e1;
break;
case 2 : compute-e2;
break;
case 3 : compute-e3;
break;
.....
}
	slave(i)
	{
	compute ei;
	wait(request);
	if(request)
	send(ei,0)
	}
	master()
	{
	compute expr;
	switch(expr)
	{
	case 1 : send(request, 1);
	receive(a1,i);
	.....
	}
	}

Example : Topological sorting

- In a directed acyclic graph topological sorting is used to provide an ordering of the vertices. For example consider two vertices u and v of a graph G. In this graph when a path exists between u and v, appears after u in the ordering.
- It is a requirement that the graph should have acyclic property, otherwise for an edge represented as (u,v) there would be a path from u to v and also from v to u, and therefore the ordering cannot be obtained.
- Now let's assume there are a number of tasks we need to be performed in which some of the tasks depend on the others and it is possible to do only one at one. These tasks can be organized in the dependency graph. Here one must be able to find out an ordering of the tasks based on the dependencies. Consider Fig. 2.2.3 for understanding of topological sorting.

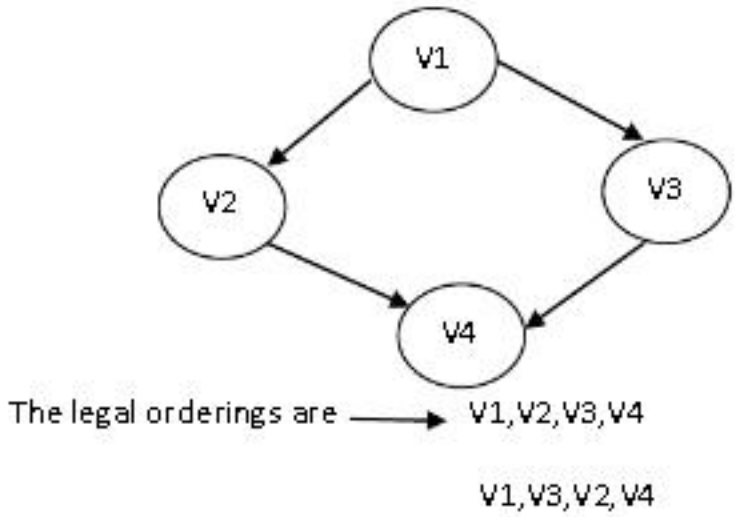


Fig. 2.2.3

- The steps of the topological sorting algorithm are given below :
  i. Initially compute the in – degrees of all the vertices in the graph.
  ii. Find U as a vertex with degree 0 and store it in the list for ordering.
  iii. At this stage is no such vertex is detected then there is a cycle found and the algorithm stops.
  iv. Remove the vertex U and all the edges (U,V) it belongs from the graph.
  v. At this step need to update in – degrees of the vertices remains.
  vi. Repeat step 2 through 4 till the vertices present for processing.
- In the topological sorting algorithm we can get the idea about the next vertex in the order before the next trace for the vertex is actually performed.

Ex. 2.2.1. : Topological sorting

Now we will discuss another example to understand topological sorting. Consider the given directed graph G=(V,E), find a linear ordering of vertices such that : for all edges(v,w) in E, v precedes w in the ordering.

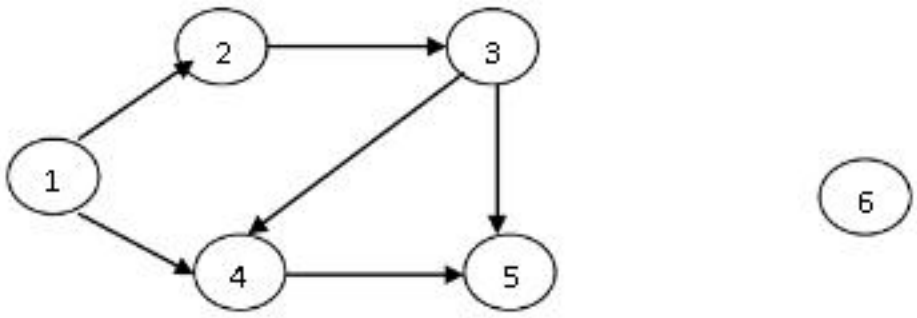


Fig. P.2.2.1

Solution : For the topological sorting any linear ordering in which all the arrows go to the right is a valid solution.

This way the linear ordering shown in Fig. P.2.2.1 (a) is a valid solution.

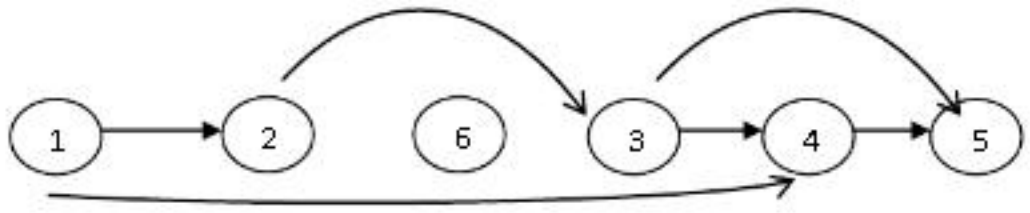


Fig. P.2.2.1 (a) : Valid topological sorted graph

The graph shown in Fig P.2.2.1 (b) is not valid topological sorted order.

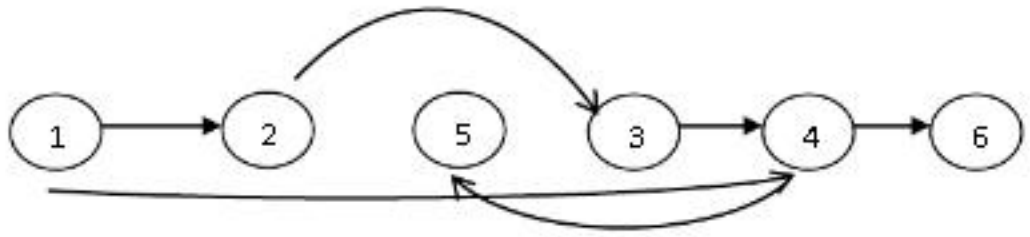


Fig. P.2.2.1 (b) : Invalid topological sorted graph

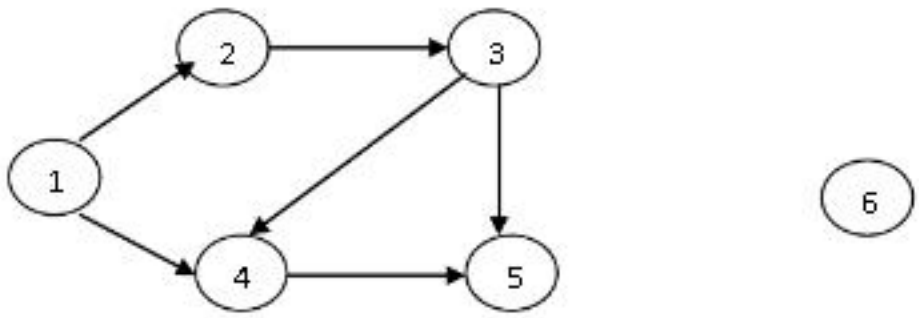
Topological sort algorithm : The steps of the topological sorting algorithm are given below.

Step 1 : Identify vertices that have no incoming edges (select one vertex)

Step 2 : Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.

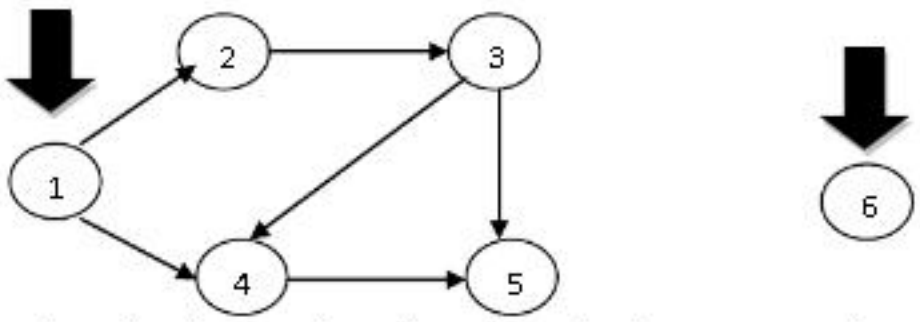
Step 3 : Repeat steps 1 and 2 until graph is empty.

Now the steps of the algorithm are performed in the given graph.



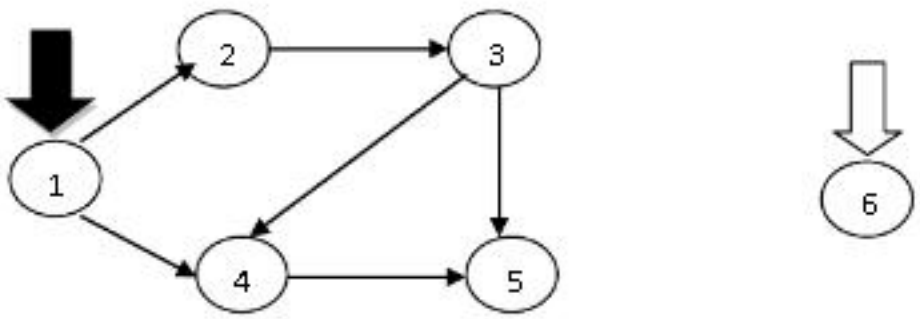
Perform Step 1 : Identify vertices that have no incoming edges. This means select vertices with in-degree zero.

Select

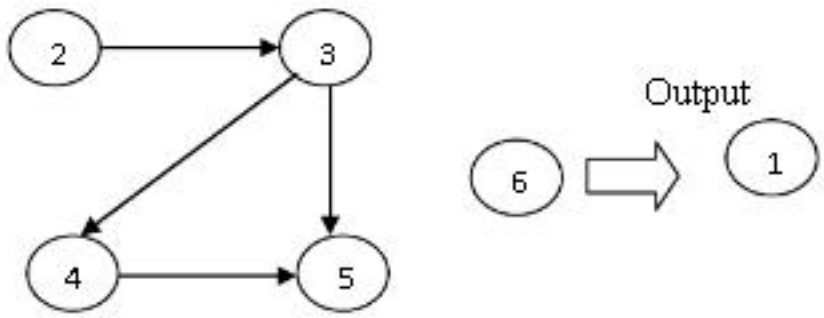


Perform Step 1 : Identify vertices that have no incoming edges. (Select one vertex).

Select

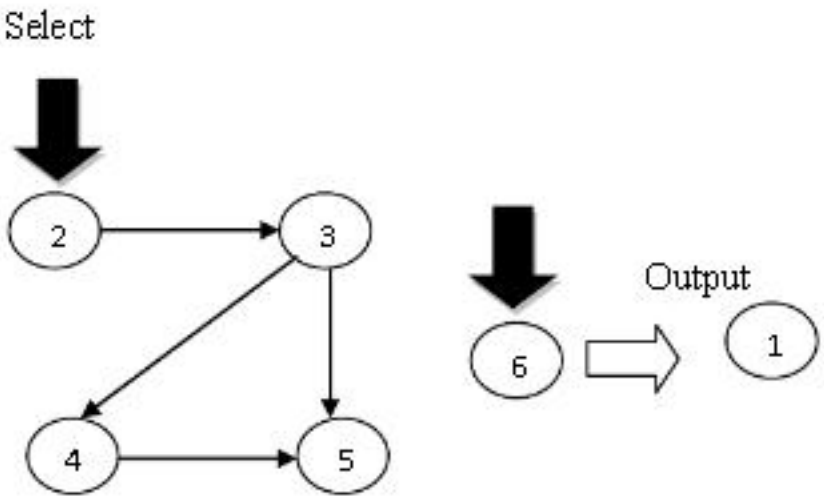


Perform Step 2 : Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.

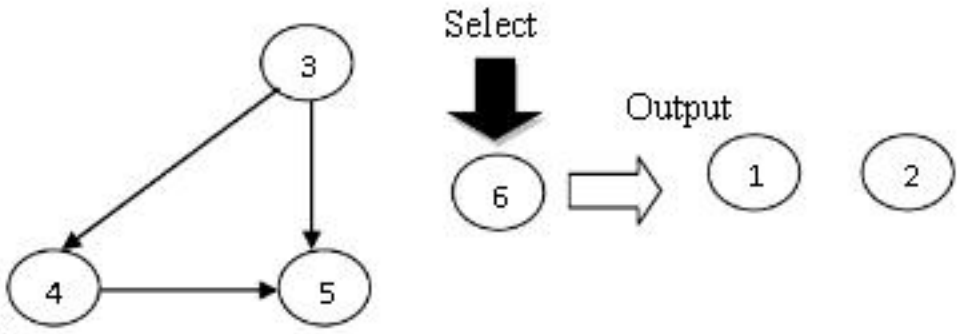


Repeat Steps 1 and Step 2 until graph is empty.

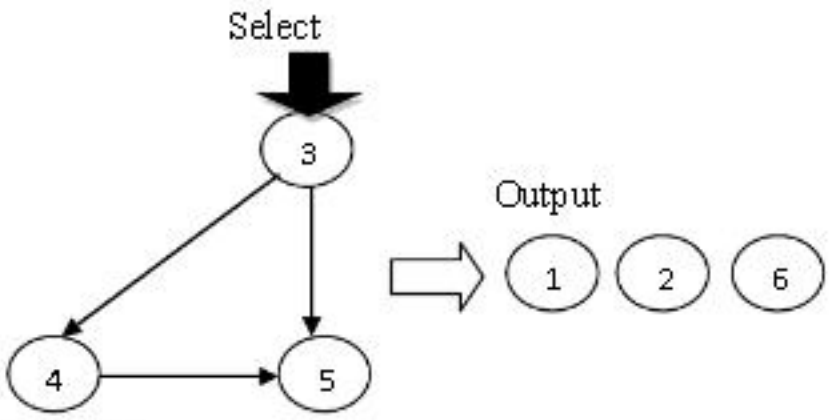
Select



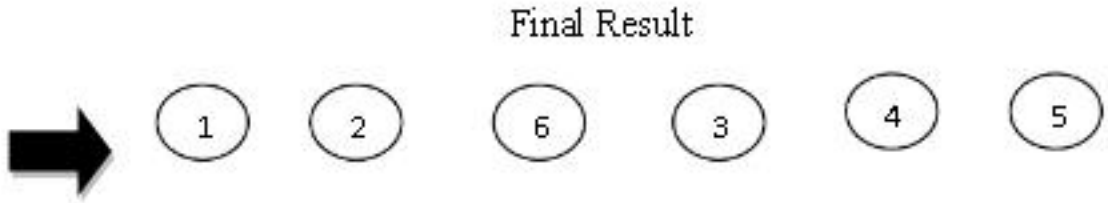
Repeat Steps 1 and Step 2 until graph is empty.



Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty



Finally the result of the topological sorting is shown here.

2.3 Characteristics of Tasks and Interactions

- The different decomposition techniques are used to divide the overall problem into several concurrent tasks so that they can be executed in parallel. The further step of the parallel algorithm design deals with the mapping of these tasks into available processes.
- The decomposition approach provides the possibilities of writing good parallel programs to be performed in efficient time. The communication amongst tasks play very crucial roles while mapping the tasks into processes for parallel executions.
- This part describes properties of tasks and inter-task communications among them for good mapping for better performance oriented parallel algorithm designs.

2.3.1 Tasks Characteristics

There are some key characteristics used to influence the choice of mapping of tasks onto available processes and at the same time the performance of the parallel algorithm is also

maintained. The four basic characteristics are Generation of task, Size of task, Knowledge of Task Sizes and Data size associated with Tasks.

Generation of Task

As we know an algorithm consists of tasks and in a parallel algorithms these tasks are considered to be executed in parallel. The tasks included in a parallel algorithm are possible to generate either on a prior basis termed as static or created whenever required termed as dynamically.

Static task generation.

- The approach in which the work(s) to be performed by an algorithm is known in advance on priority basis is called as static task generation. The tasks are defined before starting the execution of the algorithm and the specified order is to be followed by the algorithm in execution.
- For example consider the matrix multiplication algorithm where overall matrices divided into different chunks and distributed to various processors in a parallel machine. In each processor, the task for matrix multiplication executes on different sets of data considered for operation.

Dynamic task generation

- The tasks to be performed are created dynamically based upon the decomposition of the data in certain situations. In these cases the tasks to be performed are not available before algorithm executions.
- The recursive and exploratory decomposition techniques are considered as examples of dynamic task generation.

Size of tasks

- Every task takes some amount of time for its completion. The size of a task is represented by the time required for its completion. There are two different categories in which programming tasks can be divided : uniform and non-uniform.
- The tasks are required to map into available processes so that it can be computed and it depends upon the type of the tasks. There are different mapping schemes and uniform tasks tales almost same amounts of time in mapping. In the case of non – uniform tasks, the mapping times required varies in different schemes.

Knowledge of Task Sizes

- The task size knowledge is used for the choice of mapping scheme. The tasks are mapped into processes and the prior knowledge of the tasks used in mapping. As an example the

knowledge of the computation time required for each task is used to apply different decomposition techniques.

- In the similar fashion in some of the examples where dynamic decision are required during processing the prior knowledge of the task size is not known.

Data Size

The data size is one more crucial property associated with the task. The required data for performing a task should be made available when mapping it into processes. The overheads associated with the data movement can be reduced when the size of the data as well as its memory location are available.

2.3.2 Characteristics of Inter-Task Interactions

- As we know a parallel algorithm contains many tasks and these tasks may have communications with each other to share some information. The exchange of information amongst tasks is done by the use of mechanism called as inter-tasks communications.

The interactions amongst the tasks depend upon the types of interactions in different algorithms. The different programming paradigms and mapping schemes may support different natures of interaction schemes. This indicates that some of the interactions are suited to some techniques and other works most efficient with other techniques.

2.3.3 Static Versus Dynamic Pattern

There are two classes of interactions amongst the concurrent tasks based upon the types of pattern used in interactions. These patterns are static and dynamic patterns.

- **Static interaction pattern :** The static pattern specifies that the time at which interaction among tasks occur is predetermined. The set of tasks to have interactions with each other is also known before the execution of algorithms in static pattern. The message – passing paradigm like MPI(Message Passing Interface) and PVM(Parallel Virtual Machine) can be easily used to create applications with static interaction of tasks. The message – passing approach provides the specification for having association of sender and receiver tasks in message exchange.
- **Dynamic interaction pattern :** The interaction times of the tasks to be interacted as well as a set of tasks to have interactions decided at the algorithm execution time only in dynamic interaction pattern. It is always a daunting task to program dynamic interaction of tasks because of unpredictable nature of dynamic communications. The synchronization between a sender and a receiver is one of the challenging issues. This may create problem if not handled properly, particularly in a situation where both sender and receiver communicate at the same time.

2.3.4 Regular Versus Irregular

- The interactions are also places in different classes based upon their spatial structure. The instructions are kept in regular and irregular classes in consideration with spatial structures,
- The interaction pattern is considered as regular if it is possible to exploit some of the structure of the interaction pattern for efficient implementation. In the case where no such pattern exist called as irregular pattern. Like dynamic pattern of interactions the irregular pattern is also difficult to implement particularly in message – passing environment.

2.3.5 Read-only Versus Write-only

- Inter-task communication commonly called as task – interaction is used for data sharing among tasks. The choice of mapping of tasks into processes is affected by the type of sharing data between tasks.
- There are two different categories of data sharing can be considered in task – interactions : read-only and read-write.
- **Read-only interactions :** this is used in a situation where the data to be shared among concurrent tasks is required for read only purpose. The interactions among tasks in such situations are read-only interaction. Consider an array A with 10000 elements and these elements are added in parallel. In the example A is divided into multiple parts and the elements in each parallel. In this example A is divided into multiple parts and the elements in each part are added concurrently. Here the input array is required for reading purpose only. In this example the tasks require to read array A for input purpose only.
- **Read-write interactions :** this is required in a situation where many tasks have to perform read and write operations on some shared data.

2.3.6 One-way Versus Two-way

- The communications among tasks can also be categorized in terms of one way and two way communications. Suppose there is a pair of task in which one is the producer and another is the consumer.
- The interaction between producer and consumer task is termed as two-way because the data required by the consumer task is supplied by the producer task. The interaction between two tasks is typically called as two-way interaction when data required by one task is supplied by another task explicitly.
- The interaction is termed as one-way interaction when one task in a pair of tasks starts the communication and completes the interaction without interrupting other task.
- The read-only interaction in any form is considered as one-way whereas all read-write interactions can be considered as one-way or two-way depending upon the context.

Mapping

- In parallel program design it is required to specify where each task is to execute. As we have discussed in the previous section that a process performs a task. In this context the approach used for assignment of tasks to processes is called mapping.
- In fact mapping deals with two goals and these are referred as maximization of processor utilization and minimization of communication costs amongst processes.

2.1.5 Processes Versus Processors

- A program in execution is called as a process and in the design of parallel algorithms it is considered as the entity responsible for performing tasks. The computations defined to be performed in the tasks are executed by the physical unit called as processor. In this chapter the processes are used to describe parallel algorithms and programs.
- In the field of parallel computing it is assumed that the number of processors depends upon the total number of processors exit and most of the time it is considered as each process executed in a separate processor.
- In the parallel algorithm design if the algorithm is very complex and requires to have many processes so it is assumed to have running in the form of pseudo parallel processing so that the availability of processors should not resist in design.

2.4 Mapping Techniques for Load Balancing

- The mapping techniques are used to map tasks into processes so that parallel executions can be performed. The overall computation associated in a problem to be solved divided into number of tasks. These tasks are mapped onto processes with the goal of completing executions in the minimum amount of time.
- This goal can only be achieved when the overheads associated with the parallel executions of tasks are minimized. There are two factors involved with the overheads in the parallel executions of tasks. The first factor due to which overhead occurs is the total time required in inter-process interaction during communication.
- One more factor responsible for overhead is the amount of time some of the processes wait without doing any significant works typically called as idleness. These factors provide the objectives for considering mapping as a good mapping.
- The mapping is assumed as a good mapping when the computations and interactions are well balanced at each stage of the parallel algorithm.

2.4.1 Mapping Techniques

There are two different categories of mapping techniques are used in parallel algorithms : static and dynamic. The suitability of these two mapping techniques is decided on the basis of tasks characteristics and how they interact with each other.

- Static mapping : In this technique, the mapping of tasks onto processes is performed before execution of algorithms. This indicates the tasks are distributed among available processes prior to execution of algorithms. The factors like knowledge of the task size, data size and inter-task interaction characteristics are used in deciding about the mapping technique to be used. For example, these two techniques static and dynamic mapping can be used when tasks are generated statically but which mapping will perform better is decided on the basis of task factors and parallel programming paradigm.
- Dynamic mapping : In this technique, the mapping of tasks onto processes is performed during the execution of algorithms. This indicates the tasks are distributed among available processes when algorithm actually executes.
- There are basic situations where dynamic mapping is applied. The first and straightforward case is if tasks are generated dynamically then mapping technique is used. Dynamic technique is applied in a situation where the large amount of data is associated with tasks. In this situation dynamic mapping moves data among available associated with tasks. In this situation dynamic mapping moves data among available processes.

2.4.2 Schemes for Static Mapping

- The static mapping is usually used along with some partitioning techniques naturally. There are three different schemes suggested for static mapping : Mapping Based on Data Partitioning, Task Graph Partitioning and Hybrid Strategies.
- Mapping Based on Data Partitioning : The mapping is used for associating tasks into processes. One of the popular rules used in High-Performance Computing called as owner computes rule and according to this rule mapping of tasks onto processes is similar to mapping of data to processes.
- The data associated with the algorithms are commonly represented by arrays and graphs. In this section we will discuss mapping of tasks onto processes on the basis of arrays and graphs.

Arrays Distribution Schemes

- The purpose of this scheme is to distribute array elements across local memories of a parallel computer. The direct benefit of this distribution is that the element can be

accessed in parallel for processing. There are three standard distribution of the dense arrays : block, cyclic and block cyclic.

- The rule commonly used and implemented in High Performance Fortran is named as Owner Computes rule. The owner computes rule is applied in a decomposition techniques based on the partitioning of data. According to this rule mapping of tasks is similar to mapping of required data onto the processes. At this stage we will discuss commonly used techniques of distributing arrays among processes.
- The owner computes rule is most often used in High-Performance Fortran compilation systems. According to this rule the required calculation will be performed by the processor that owns the left had side element. This is described using an example loop shown below in Fortran language.

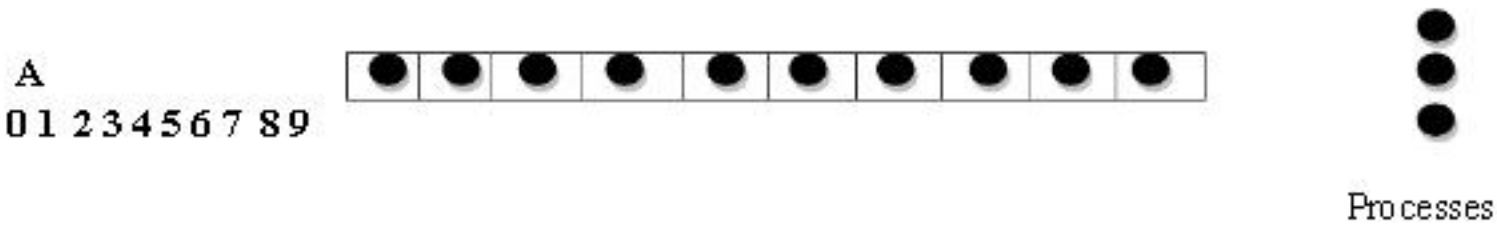
```
Do i=1, n

a(i-3) = b(i*2)/c(i+j)-a(i**i)

END Do
```

In this example the expression a(i-3) is the left hand side of the assignment statement. This expression is owned by a processor in a multiprocessor system. This induces that the assignment will be performed by the processor owns a(i-3) expression. Therefore all the components present at the right hand side should be made available to the processor for performing computation without any hassle.

Block Distributions

- The block distribution deals with the homogenous distribution of computational load over a regular data structure such as a Cartesian grid. According to block distribution blocks of size S of the vector is assigned to processes for mapping of elements during processing.
- This distribution is based on the approach of assigning continuous portions of the array to different processes. In this distribution technique an array with m – dimension is distributed among a set of processes in the fashion such as each process gets contiguous block of array element.
- The block distribution of array elements is most of the time suitable when the computation perforrms on the elements of an array called as locality of interaction.
- The block distribution is described in Fig 2.4.1 with pictorial representation. In Fig 2.4.1, an array named as A of size 10 is considered. Also assume four processes p0,p1,p2, and p3 execute in parallel to compute on array elements.
- In the first case the array elements area distributed to four processes. Here, the block size is assumed three. This means processes P0, P1, and P2 will get three elements each of the array A whereas process P3 will get remaining one element.

P0
P1
P2
P3



Distribute array elements into processes with block size as three :

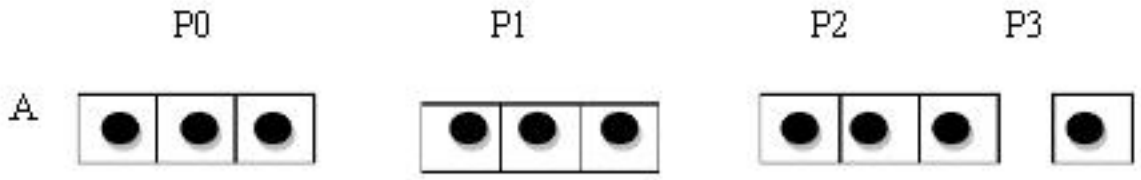


Fig 2.4.1

- In the first case, the array elements are distributed to four processes. Here, in Fig 2.4.2, the block size is assumed four. This means processes P0, P1 will get four elements have been distributed. The last process P3 will not get any element.

Distribute array elements into processes with block size as four :

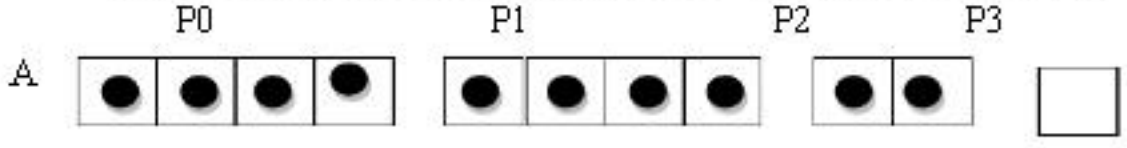


Fig 2.4.2

- In this scheme, the A[i] element is mapped to the processor[i/b] if distribution is BLOCK(b).
- Now, we will consider the distribution of an array A of M elements over P processors. The mapping of the global index m where it lies between 0 to M represented as (0 <= m < M) is handled. The global index m of the data object will be mapped to an index pair (P,I).
- In the current discussion p is used to specify processes to which the elements are mapped. The value of p is represented by 0 <= p < P with indication of its range. The value of variable I indicated the location of the element in the array. The mapping represented as m⟶(p,i) for block distribution is defined as :

m→(oor(m/L),m mod L)

Where, L is defined as L = ceiling (M/P).

- The block distribution is described using as example where the parameters p, m etc are used. Let's assume we have M = 23 data elements to be distributed over three processes (P = 3). Also consider the block size S = 8.

m	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22
p	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	2 2 2 2 2 2 2
i	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
B	0 0 0 0 0 0 0	1 1 1 1 1 1 1	2 2 2 2 2 2 2

- It is clear in the example that uneven distribution occurs. Here because of uneven distribution the last block becomes smaller than the others. A global block number B is also shown here for indication of block distribution.
- Distribution of M X N matrix using block distribution : In block distribution of any array each process gets a contiguous part of the array data. An M X NX matrix is distributed using two different approaches : row-wise and column-wise distribution. Each process receives N/P rows of a matrix; here P indicates the number of processes used for mapping of tasks when elements are distributed along with row-wise distribution approach. On the other hand each process gets M/P columns of the matrix when column-wise distribution approach is applied. The block distribution approaches is suitable for some computations like matrix-vector multiplication.
- 2-D distribution on process : We have to assume a process grid for the distribution of array elements on processes. In general one has to assume the size of the process grid, assume now as P1 X P2. The number of processes for which this grid can be used is P = P1 * P2. For example consider the distribution of N X N matrix. The matrix can be divided into N/P1 X N/P2 sub matrices. Fig 2.4.3 shows the 4 X 4 process grid.

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Fig. 2.4.3

- This 4 X 4 process grid is used to execute processes in parallel with distribution of data in the form of the rid. Once the data required by a process is provided that process can start execution. In this way all processes operate on their part of the data concurrently.

2.4.3 Cyclic Distributions

- In the situation where computational load is distributed in inhomogeneous fashion, cyclic distribution is used. The cyclic distribution is used in such case to improve load balancing.
- The consecutive entries of the global vector are used to assign in successive processes. The cyclic distribution of data among processes does not use any concept of block numbers. The mapping m $\rightarrow$ (p,i) is defined as shown below for cyclic distribution :

m→(m mod P, oor(m/P))

- This scheme of distribution can be used to reduce the problem like the load-imbalance and idling. The load imbalance occur when the amount of work is different for different part of a matrix. This can be avoided by using the cyclic or block cyclic distributions.
- The cyclic distribution is described in Fig. 2.4.4 using an example where the parameters p, m etc. are used. Let's assume we have M = 23 data elements to be distributed over three processes (p=3).

m	0 1 2	3 4 5	6 7 8	9 10 11	12 13 15	16 17 18	18 19 20	21 22
p	0 1 2	0 1 2	0 1 2	0 1 2	0 1 2	0 1 2	0 1 2	0 1
i	0 0 0	1 1 1	2 2 2	3 3 3	4 4 4	5 5 5	6 6 6	7 7

Fig. 2.4.4

- **Example : 1D Cyclic distribution of array 'A' on four processes :** The cyclic distribution is described using an example shown in Fig 2.4.5. Assume there are four processes P0, P1, P2 and P3 are available so that elements of an array can be mapped onto these for doing computations.

Processes : P0, P1, P2, P3

Fig 2.4.5 (a)

Now the cyclic distribution of array 'A' becomes as show in fig 2.4.5(a)

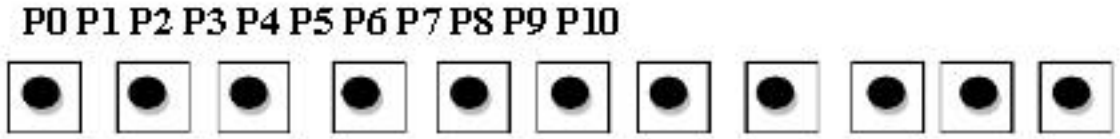


Fig. 2.4.5 (b)

- In Fig 2.4.6 the distribution of elements to processes are shown according to cyclic distribution approach. In this example 1-D cyclic distribution on four processes is described.

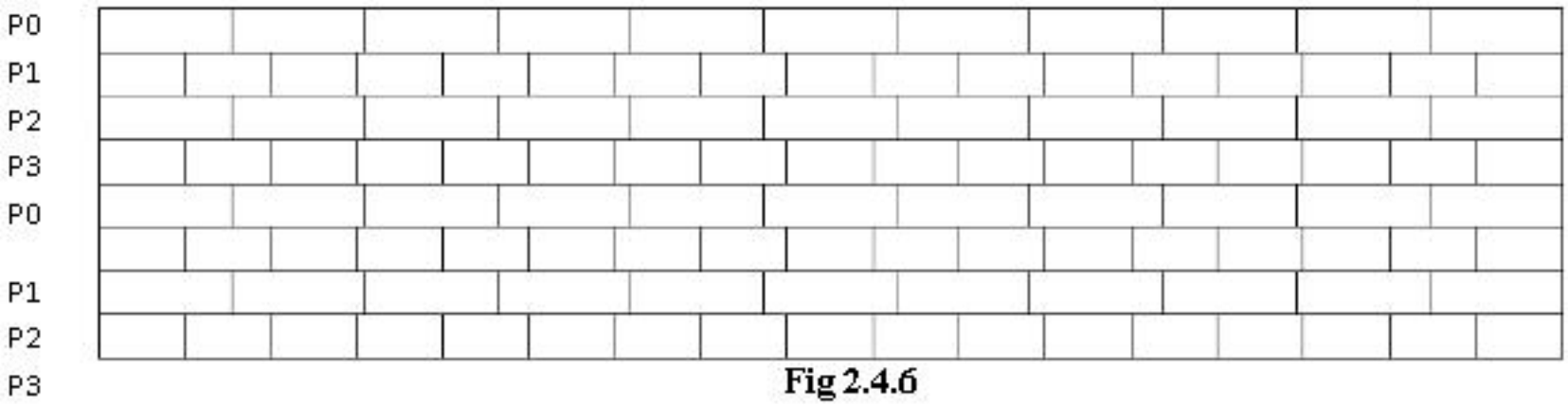


Fig 2.4.6

2.4.4 Block – Cyclic Distribution

- This type of data distribution onto processes is another technique which is slight variation of the block distribution. The block-cyclic distribution is the generalization of the block and cyclic distribution.
- In this form of distribution a block is considered which consists of number of consecutive data objects represented by r. These consecutive data objects are distributed cyclically onto number of processes represented by p.
- The parameters used here are the global index m and index triplet (p,b,i). In this case the global index m lies between 0 and M and represented as m(0<=m< M). In the index triplet (p,b,i) the value of p lies between 0 and p and represented as p(0<=p<P).
- The global index of the data object is mapped into index triplet where p specifies processes onto which elements are mapped. The triplet (p,b,i) has p to be specified the process to which data element is mapped. The block number is represented by in the process p and i is the location in the block. The mapping m → (p,b,i) in block – cyclic distribution is defined as shown below.

m→(oor((m mod T)/r), oor(m/T), m mod r)

- It is require to know about the block – cyclic distribution is that it reverts to the cyclic distribution in which r=1 and a block distribution where r = L. Here T = rP.
- Consider an example of block – cyclic distribution where M = 23 data objects. These data objects are distributed over three processes(p=3). The block size represented by r is 2.
- The uneven distribution of data objects are shown in Fig 2.4.7. This happens because the last block is smaller that others.

m	0 1 2 3 4 5	6 7 8 9 10 11	12 13 14 15 16 17	18 19 20 21 22
p	0 0 1 1 2 2	0 0 1 1 2 2	0 0 1 1 2 2	0 0 1 1 2
b	0 0 0 0 0 0	1 1 1 1 1 1	2 2 2 2 2 2	3 3 3 3 3
i	0 1 0 1 0 1	0 1 0 1 0 1	0 1 0 1 0 1	0 1 0 1 0
B	0 0 1 1 2 2	3 3 4 4 5 5	6 6 7 7 8 8	9 9 10 10 11

Fig 2.4.7

2.4.5 Randomized Block Distributions

- This type of distribution is considered as a more general form of block distribution. The load balancing is achieved by dividing the array into more number of blocks than the total number of processes currently active for proceeding the data.
- The load balancing in randomized distribution is similar to a block – cyclic distribution. In this technique blocks are distributed randomly among processes but the block distribution is handled uniformly.
- Now consider an example of a one dimensional block distribution in randomized fashion shown in fig 2.4.8
- The vector V with 15 elements included is distribution using randomized block distribution. The assumed block size is 3 and five processes are available.

V = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14}

Random(V) = {11,5,13,8,14,1,6,7,3,10,0,12,2,4,9}

Now perform mapping of blocks onto processes.

{11,5,13, 8,14,1, 6,7,3 10,0,12 2,4,9}

Fig. 2.4.8

2.4.6 Hierarchical Mappings

- There are some algorithms can be represented using Task dependency graphs. They are represented in the straight forward manner because of the task implemented on those are naturally suited to represent using Task dependency graphs. The mapping in this approach also has some problems like load imbalance or inadequate concurrency.
- For example hierarchical mapping can be described for the binary tree task dependency graph. One such binary tree is shown in Fig. 2.4.9. Here at the top of the tree only few concurrent tasks can be performed because of availability of less number of tasks.
- The decomposition is expected at further level when the level at which number of tasks available are large so that mapping can be handled efficiently. In such cases a large task is divided into smaller sub – tasks at further levels.

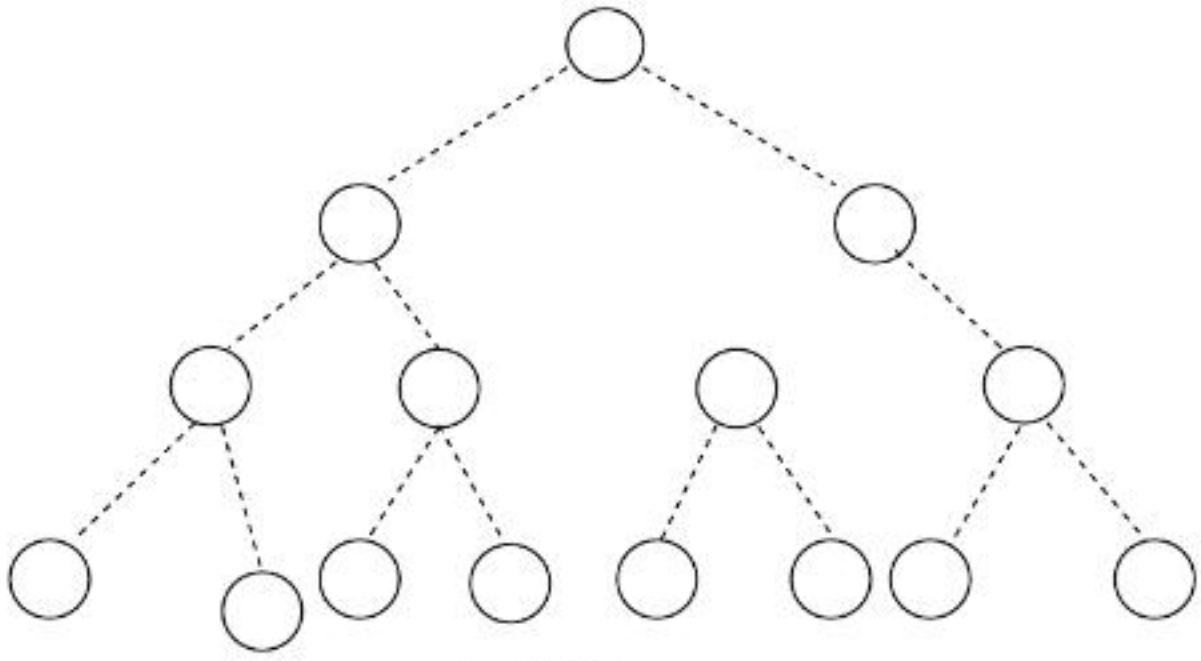


Fig 2.4.9

- In Fig. 2.4.9, the binary tree is shown with level size as 4. Here initially the root task can be divided and assigned into eight processes. In the similar fashion task at the level next to root can be divided and assigned into four processes.
- These divisions at different levels are used for mapping of tasks onto processes and finally at leaf level tasks are mapped with processes in the form of one to one mapping. The following pictorial representation shows the hierarchical mapping based on the descriptions we have just made.

2.4.7 Schemes for Dynamic Mapping

- Dynamic mapping is applied in the situation where static mapping is not efficient because it shows very highly imbalanced distribution of works. The dynamic mapping of work distribution to processes used in another situation where task-dependency graph itself is dynamic by nature.
- The dynamic mapping is also called as dynamic load – balancing because it is basically used for balancing of workloads among processes. There are two different classes of dynamic mapping centralized and distribution.

Dynamic mapping with Centralized Schemes

- In this scheme a common data structure is used centrally to maintain all executables tasks. These tasks can also be maintained centrally by a special process. The processes involved in such a system are managed in the master – slave fashion. The master process is a special process responsible to manage the pool of available tasks.
- The other processes referred as slave processes get their work assigned from the master process. Any newly generated tasks are stored in the centralized data structure and master process assigns tasks from this structure to slave processes.

- The main advantage of the centralized scheme is its simplicity in implementation. It is really easier in terms of implementation as compare to distributed scheme. The limited scalability is the disadvantage of this scheme.
- For example, in the case when number of processes has to access central data structure increases; the efficiency of the master process of assign tasks may degrade the overall performance.
- Consider sorting of an example matrix A of size M X N using the centralized mapping approach. The program segment shown below is used to do sorting of entries in each row of A in serial order.

```
for(i=0;i<M;i++)

sort(A[i], M)
```

- The sorting of initial elements of an array affects the total sorting time of a particular sorting algorithm. This way the time required for sorting of an array elements vary because of dependency of sorting of initial elements.
- This concludes that each iterations of the 'for loop' used in the program segment considered in this example may take different amount of time.
- In such type of situations load – imbalance may occur when mapping of the task (sorting of an equal number of rows) onto processes. There are two techniques have been suggested for handling load-imbalance problem are : self-scheduling and chunk scheduling.
- **Self-Scheduling** : This method is used for scheduling of independent iterations of a loop on number of available processes in parallel. This way it may happen that separate iteration of the loop is assigned to different processes for executions.
- All such processes then execute iterations of the loop in the parallel fashion. The approach is based on the use of a central pool of row indices of not yet sorted elements.
- When a process becomes available it picks up an index from the pool and performs sorting on that row. After sorting successfully that index is deleted from the pool of indices. This way all rows are sorted till the pool becomes empty.
- **Chunk scheduling :** The overall task is divided into number of subtasks. A chunk refers to a group of such tasks. In this scheme chunks are created and assigned to different processes for parallel executions. The problem of load-imbalance occurs when the number of tasks assigned in a single step is large.
- The problem becomes more severe when the chunk size increases during program progress. This problem is solved by reducing the chunk size when program progresses further. This means when the program execution starts the chunk size remains large but when program further progresses the chunk size also reduce.
-

Dynamic mapping with Distributed Schemes

- In this scheme the set of available executable tasks are distributed among processes for executions. The work balance is handled at runtime with exchange of tasks among processes.
- In this scheme it is assumed that communications among processes for sending and receiving work-loads is possible. The main advantage of this scheme is that these methods do not encounter any type of bottleneck.

2.5 Methods for Containing Interaction Overheads

- A parallel program performs efficiently when the interactions among the concurrent tasks are efficiently handled. There are many factors associated due to which interaction overhead increases in concurrent tasks.
- Some of the factors which are most important and require attention are: amount of data exchanged during interactions, the frequency of interactions and the spatial and temporal pattern of interactions.
- The general techniques used to reduce the interactions overheads are applicable during the stages when the decomposition and mapping schemes for the algorithms are devised. Some techniques are useful algorithm is being programmed in a given paradigm.

2.5.1 Maximizing Data Locality

- The tasks executed by different processes have to access some common data in most of the parallel programs. A particular task is able to compute efficiently when the required data upon which it operates are available in time. The data locality technique is used here to make the required data available to the tasks so that task interactions can be reduced.
- The various techniques like increase the reuse of recently accessed data and reduce the data access frequency up to some level so that data locality can be maximized. The techniques used in the modern processors for improvement of the data availability through cache, work almost similar like the schemes implemented for improving the data availability.
- **Minimize Volume of Data-Exchange:** The interaction overhead is reduced using the basic technique in which the overall volume of shared data is minimized. In this basic approach the shared data that needs to be accessed by concurrent tasks are targeted to be minimized.
- This approach is based on the facility where the locality of the temporal data is maximized. This way the consecutive references to the same data increases. The requirements for bringing more data into local memory of cache are minimized because of performing most of the computations on local data.

- One more approach is considerable here to decrease the amount of shared data required to be accessed by multiple processes is use of local data for intermediate results. This way shared data access is performed only for storing of the final results of the computations.
- **Minimize Frequency of Interactions:** The overheads occur in the interaction of tasks in parallel programming is reduced by minimizing the frequency of interaction. The use of shared data in large pieces through restricting of algorithm may affect the reduction in interaction frequency. This way the overall interaction overhead is reduced, even if such restricting does not necessarily reduce the overall volume of shared data that need to be accessed.

2.5.2 Overlapping Computations with Interactions

- In the parallel executions different processes involved in the computations have to cooperate and communicate with each other. These processes have to wait shared data to be arrived from other processes.
- The overlapping of computations with the interaction van be handled when a process has to wait for interaction at that time it can be assigned to do other independent computation. There are various approaches used to overlap computations with interactions.
- We can consider an example where overlapping is possible, simply identify the parts of the program code can be executed before initiating the interaction. Thereafter the parallel program is required to be structured in such a way that the interaction is to be initiated at an earlier point in the execution than it is needed in the original algorithm.

2.5.3 Replicating Data or Computations

- The interaction overheads amongst the concurrent tasks can also be reduced using the replication of data and computations.
- The explicit programmer intervention is not required for the read – only data needed for frequent access in the shared – address – space paradigm because such data very often affected by the caches.
- The architecture and programming paradigm where read-only access to shared data is expensive and harder as compare to local data access the obvious solution is based on replication of the data. This way parallel programming using message – passing paradigm benefits from the replication of data.
- The message – passing significantly reduces the interaction overheads and simplifiers the programming.
- The downside of the data replication is increased memory requirements of a parallel program. The overall memory requirement for replicated data storage increases linearly with the number of concurrent processes.

- The direct restriction imposed because of the memory limitation is the limitations on the size of the problem to be solved on a particular parallel machine.

2.5.4 Overlapping Interactions with Other Interactions

- The overlapping of interaction with other interaction amongst several pairs of processes can reduce the effective volume of communications. We can consider four processes P0, P1, P2, and P3 as an example of overlapping interaction in a message – passing paradigm.
- In this paradigm broadcasting of data from process P0 to all other processes works in the fashion as follows. Process P0 sends data to be sent to process P2. In the next step the data is transferred to process P1 and concurrently P2 sends the data it had received from P0 to P3. In this way the operation is completed in two steps instead of multiple steps.

2.6 Parallel Algorithm Models

- Having discussed the techniques for decomposition, mapping, and minimizing interaction overheads, we now present some of the commonly used parallel algorithm models. An algorithm model is typically a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- **2.6.1 The Data-Parallel Model**
- The data-parallel model is one of the simplest algorithm models. In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms.

- Interaction overheads in the data-parallel model can be minimized by choosing a locality preserving decomposition and, if applicable, by overlapping computation and interaction and by using optimized collective interaction routines. A key characteristic of data-parallel problems is that for most problems, the degree of data parallelism increases with the size of the problem, making it possible to use more processes to effectively solve larger problems.
- An example of a data-parallel algorithm is dense matrix multiplication described in Section 3.1.1. In the decomposition shown in Figure 3.10, all tasks are identical; they are applied to different data.

- **2.6.2 The Task Graph Model**
  As discussed in Section 2.1, the computations in any parallel algorithm can be viewed as a task-dependency graph. The task-dependency graph may be either trivial, as in the case of matrix multiplication, or nontrivial (Problem 3.5Typical interaction-reducing techniques applicable to this model include reducing the volume and frequency of interaction by promoting locality while mapping the tasks based on the interaction pattern of tasks, and using asynchronous interaction methods to overlap the interaction with computation.

  Examples of algorithms based on the task graph model include parallel quicksort (Section 9.4.1), sparse matrix factorization, and many parallel algorithms derived via divide-and-

conquer decomposition. This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called task parallelism.

- **2.6.3 The Work Pool Model**
- The work pool or the task pool model is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process. There is no desired premapping of tasks onto processes.

- In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks. As a result, tasks can be readily moved around without causing too much data interaction overhead. The granularity of the tasks can be adjusted to attain the desired level of tradeoff between load-imbalance and the overhead of accessing the work pool for adding and extracting tasks.

- Parallelization of loops by chunk scheduling (Section 3.4.2) or related methods is an example of the use of the work pool model with centralized mapping when the tasks are statically available. Parallel tree search where the work is represented by a centralized or distributed data structure is an example of the use of the work pool model where the tasks are generated dynamically.

- **2.6.4 The Master-Slave Model**
- In the master-slave or the manager-worker model, one or more master processes generate work and allocate it to worker processes. The tasks may be allocated a priori if the manager can estimate the size of the tasks or if a random mapping can do an adequate job of load balancing. In another scenario, workers are assigned smaller pieces of work at different times. The latter scheme is preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces. In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated. In this case, the manager may cause all workers to synchronize after each phase.
- While using the master-slave model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master. It may also reduce waiting times if the nature of requests from workers is non-deterministic.

- **2.6.5 The Pipeline or Producer-Consumer Model**
- In the pipeline model, a stream of data is passed on through a succession of processes, each of which perform some task on it. This simultaneous execution of different programs on a data stream is called stream parallelism. With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline.

- Load balancing is a function of task granularity. The larger the granularity, the longer it takes to fill up the pipeline, i.e. for the trigger produced by the first process in the chain to propagate to the last process, thereby keeping some of the processes waiting. However, too fine a granularity may increase interaction overheads because processes will need to interact to receive fresh data after smaller pieces of computation. The most common interaction reduction technique applicable to this model is overlapping interaction with computation.

- **2.6.6 Hybrid Models**
- In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. In some cases, an algorithm formulation may have characteristics of more than one algorithm model. For instance, data may flow in a pipelined manner in a pattern guided by a task-dependency graph.

2.7 Parallel Processing:

Parallel processing is a method of simultaneously breaking up and running program tasks on multiple microprocessors, thereby reducing processing time. Parallel processing may be accomplished via a computer with two or more processors or via a computer network.

Parallel processing is also called parallel computing.

2.8 GPU Computing:

GPU computing is the use of a GPU (graphics processing unit) as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing.

The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU.

From a user's perspective, the application runs faster because it's using the massively parallel processing power of the GPU to boost performance. This is known as "heterogeneous" or "hybrid" computing.

A CPU consists of four to eight CPU cores, while the GPU consists of hundreds of smaller cores. Together, they operate to crunch through the data in the application. This massively parallel architecture is what gives the GPU its high compute performance.

There are a number of GPU-accelerated applications that provide an easy way to access high-performance computing (HPC).

Core comparison between a CPU and a GPU

Application developers harness the performance of the parallel GPU architecture using a parallel programming model invented by NVIDIA called "CUDA."

All NVIDIA GPUs - GeForce®, Quadro®, and Tesla® - support the NVIDIA® CUDA® parallel-programming model.

Tesla GPUs are designed as computational accelerators or companion processors optimized for scientific and technical computing applications. The latest Tesla 20-series GPUs are based on the latest implementation of the CUDA platform called the "Fermi architecture".

Fermi has key computing features such as 500+ gigaflops of IEEE standard double-precision floating-point hardware support, L1 and L2 caches, ECC memory error protection, local user-managed data caches in the form of shared memory dispersed throughout the GPU, coalesced memory accesses, and more.

History Of GPU Computing

Graphics chips started as fixed-function graphics pipelines. Over the years, these graphics chips became increasingly programmable, which led NVIDIA to introduce the first GPU.

In the 1999-2000 timeframe, computer scientists, along with researchers in fields such as medical imaging and electromagnetics, started using GPUs to accelerate a range of scientific applications. This was the advent of the movement called GPGPU, or General Purpose GPU computing.

The challenge was that GPGPU required the use of graphics programming languages like OpenGL and Cg to program the GPU. Developers had to make their scientific applications look like graphics applications and map them into problems that drew triangles and polygons.

This limited the accessibility to the tremendous performance of GPUs for science.

NVIDIA realized the potential of bringing this performance to the larger scientific community and invested in modifying the GPU to make it fully programmable for scientific applications. Plus, it added support for high-level languages like C, C++, and Fortran. This led to the CUDA parallel computing platform for the GPU.

UNIT – III Basic Communication

3.1 One-to-All Broadcast and All-to-One Reduction

Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them.

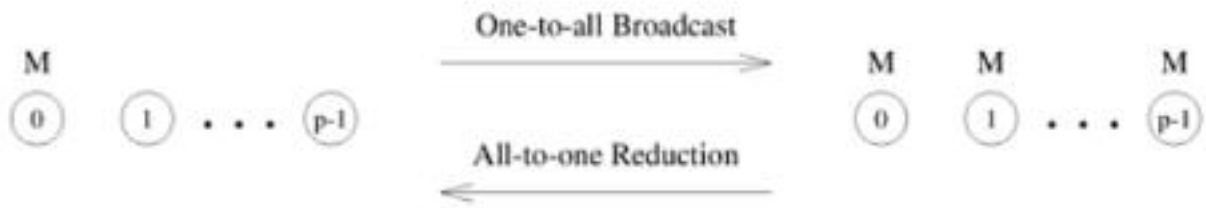
This operation is known as one-to-all broadcast. Initially, only the source process has the data of size m that needs to be broadcast.

At the termination of the procedure, there are p copies of the initial data - one belonging to each process. The dual of one-to-all broadcast is all-to-one reduction.

In an all-to-one reduction operation, each of the p participating processes starts with a buffer M containing m words. The data from all processes are combined through an associative operator and accumulated at a single destination process into one buffer of size m.

Reduction can be used to find the sum, product, maximum, or minimum of sets of numbers - the ith word of the accumulated M is the sum, product, maximum, or minimum of the i th words of each of the original buffers. Figure 3.1 shows one-to-all broadcast and all-to-one reduction among p processes.

Figure 3.1. One-to-all broadcast and all-to-one reduction.



One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety of interconnection topologies.

3.1.1 Ring or Linear Array

A naive way to perform one-to-all broadcast is to sequentially send p - 1 messages from the source to the other p - 1 processes. However, this is inefficient because the source process becomes a bottleneck.

Moreover, the communication network is underutilized because only the connection between a single pair of nodes is used at a time. A better broadcast algorithm can be devised using a technique commonly known as recursive doubling.

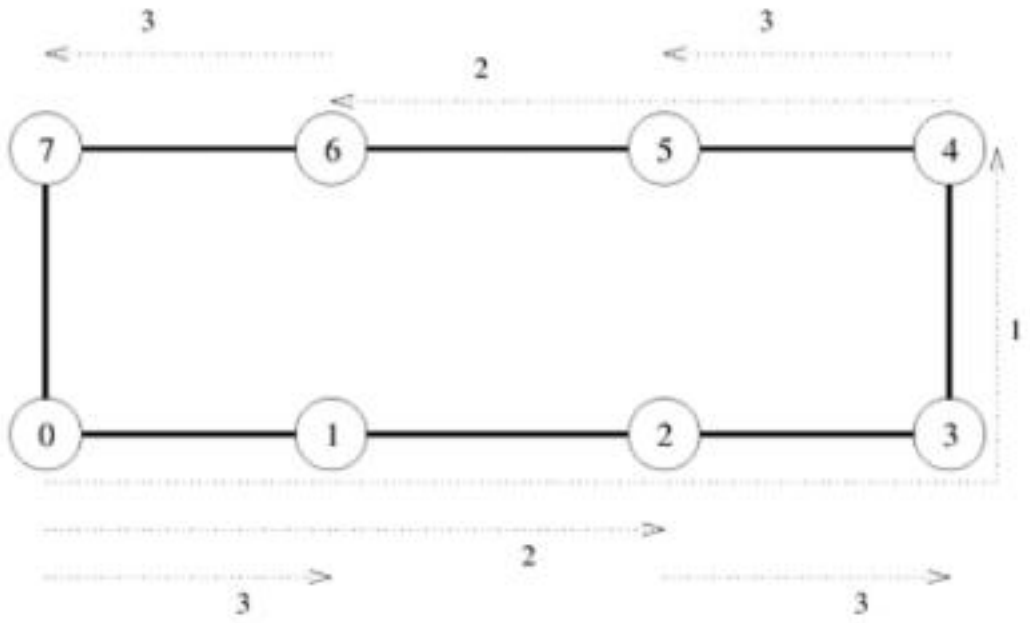
The source process first sends the message to another process. Now both these processes can simultaneously send the message to two other processes that are still waiting for the message.

By continuing this procedure until all the processes have received the data, the message can be broadcast in log p steps.

The steps in a one-to-all broadcast on an eight-node linear array or ring are shown in Figure 3.2. The nodes are labeled from 0 to 7.

Each message transmission step is shown by a numbered, dotted arrow from the source of the message to its destination. Arrows indicating messages sent during the same time step have the same number.

Figure 3.2. One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

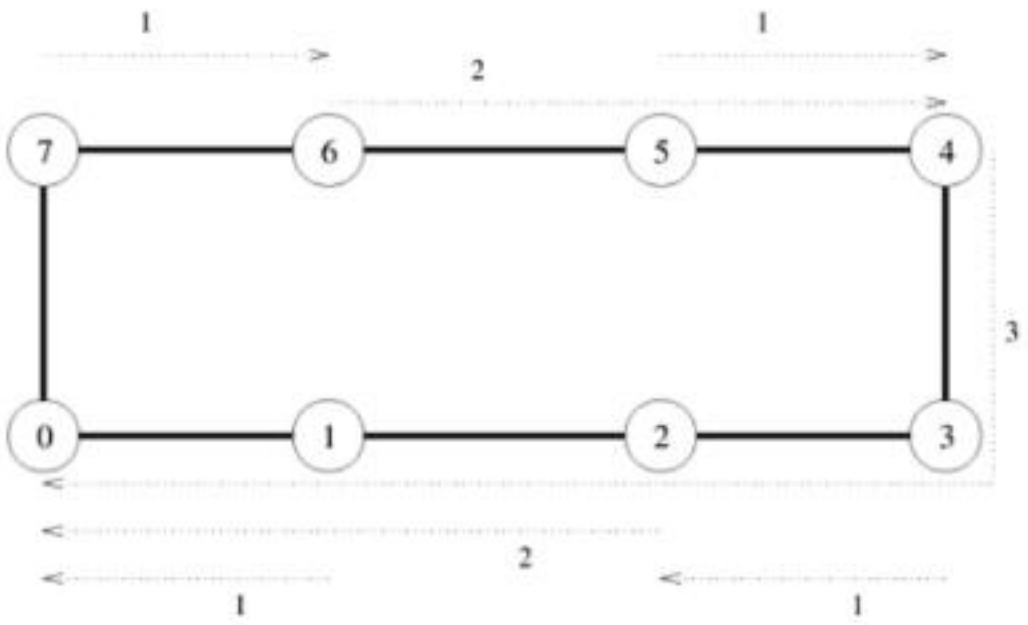


Note that on a linear array, the destination node to which the message is sent in each step must be carefully chosen. In Figure 3.2, the message is first sent to the farthest node (4) from the source (0). In the second step, the distance between the sending and receiving nodes is halved, and so on. The message recipients are selected in this manner at each step to avoid congestion on the network. For example, if node 0 sent the message to node 1 in the first step and then nodes 0 and 1 attempted to send messages to nodes 2 and 3, respectively, in the second step, the link between

nodes 1 and 2 would be congested as it would be a part of the shortest route for both the messages in the second step.

Reduction on a linear array can be performed by simply reversing the direction and the sequence of communication, as shown in Figure 3.3. In the first step, each odd numbered node sends its buffer to the even numbered node just before itself, where the contents of the two buffers are combined into one. After the first step, there are four buffers left to be reduced on nodes 0, 2, 4, and 6, respectively. In the second step, the contents of the buffers on nodes 0 and 2 are accumulated on node 0 and those on nodes 6 and 4 are accumulated on node 4. Finally, node 4 sends its buffer to node 0, which computes the final result of the reduction.

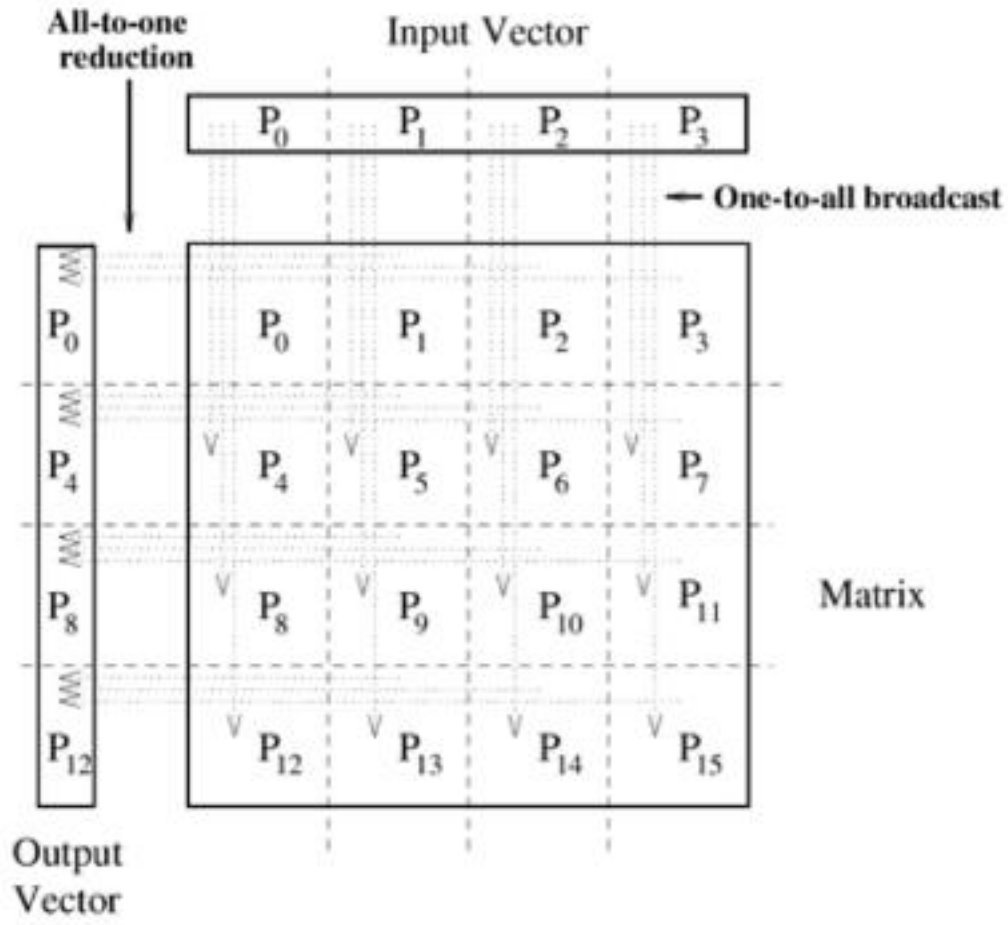
Figure 3.3. Reduction on an eight-node ring with node 0 as the destination of the reduction.



Example 3.1 Matrix-vector multiplication

Consider the problem of multiplying an n x n matrix A with an n x 1 vector x on an n x n mesh of nodes to yield an n x 1 result vector y. Algorithm 8.1 shows a serial algorithm for this problem. Figure 3.4 shows one possible mapping of the matrix and the vectors in which each element of the matrix belongs to a different process, and the vector is distributed among the processes in the topmost row of the mesh and the result vector is generated on the leftmost column of processes.

Figure 3.4. One-to-all broadcast and all-to-one reduction in the multiplication of a 4 x 4 matrix with a 4 x 1 vector.



Since all the rows of the matrix must be multiplied with the vector, each process needs the element of the vector residing in the topmost process of its column.

Hence, before computing the matrix-vector product, each column of nodes performs a one-to-all broadcast of the vector elements with the topmost process of the column as the source.

This is done by treating each column of the n x n mesh as an n-node linear array, and simultaneously applying the linear array broadcast procedure described previously to all columns.

After the broadcast, each process multiplies its matrix element with the result of the broadcast. Now, each row of processes needs to add its result to generate the corresponding element of the product vector.

This is accomplished by performing all-to-one reduction on each row of the process mesh with the first process of each row as the destination of the reduction operation.

For example, P_9 will receive x [1] from P_1 as a result of the broadcast, will multiply it with A[2, 1] and will participate in an all-to-one reduction with P_8, P_{10}, and P_{11} to accumulate y[2] on P_8.

3.2 All-to-All Broadcast and Reduction

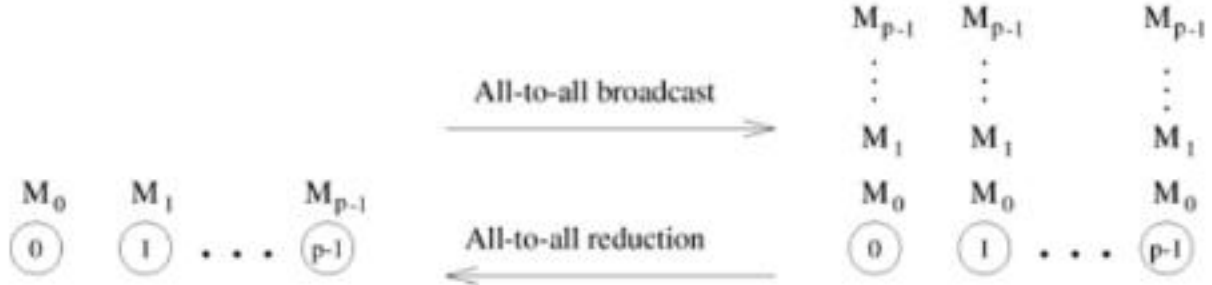
All-to-all broadcast is a generalization of one-to-all broadcast in which all p nodes simultaneously initiate a broadcast.

A process sends the same m-word message to every other process, but different processes may broadcast different messages.

All-to-all broadcast is used in matrix operations, including matrix multiplication and matrix-vector multiplication. The dual of all-to-all broadcast is all-to-all reduction, in which every node is the destination of an all-to-one reduction .

Figure 3.4 illustrates all-to-all broadcast and all-to-all reduction.

Figure 3.4. All-to-all broadcast and all-to-all reduction.



One way to perform an all-to-all broadcast is to perform p one-to-all broadcasts, one starting at each node.

If performed naively, on some architectures this approach may take up to p times as long as a one-to-all broadcast.

It is possible to use the communication links in the interconnection network more efficiently by performing all p one-to-all broadcasts simultaneously so that all messages traversing the same path at the same time are concatenated into a single message whose size is the sum of the sizes of individual messages.

The following sections describe all-to-all broadcast on linear array, mesh, and hypercube topologies.

3.2.1 Linear Array and Ring

While performing all-to-all broadcast on a linear array or a ring, all communication links can be kept busy simultaneously until the operation is complete because each node always has some information that it can pass along to its neighbor.

Each node first sends to one of its neighbors the data it needs to broadcast. In subsequent steps, it forwards the data received from one of its neighbors to its other neighbor.

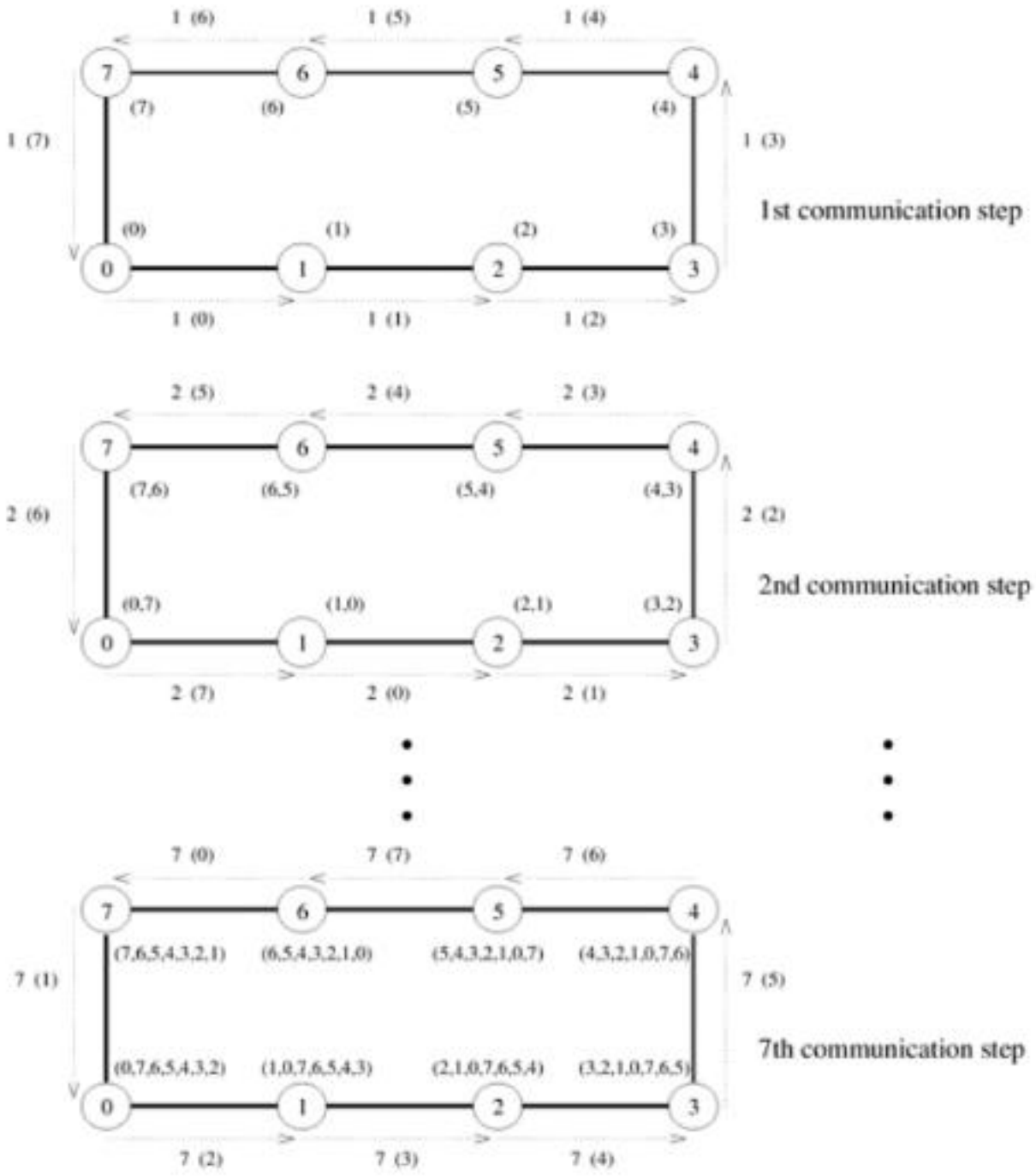
Figure 3.5 illustrates all-to-all broadcast for an eight-node ring. The same procedure would also work on a linear array with bidirectional links. As with the previous figures, the integer label of an arrow indicates the time step during which the message is sent.

In all-to-all broadcast, p different messages circulate in the p-node ensemble. In Figure 3.5, each message is identified by its initial source, whose label appears in parentheses along with the time step.

For instance, the arc labeled 2 (7) between nodes 0 and 1 represents the data communicated in time step 2 that node 0 received from node 7 in the preceding step.

As Figure 3.5 shows, if communication is performed circularly in a single direction, then each node receives all (p - 1) pieces of information from all other nodes in (p - 1) steps.

Figure 3.5. All-to-all broadcast on an eight-node ring. The label of each arrow shows the time step and, within parentheses, the label of the node that owned the current message being transferred before the beginning of the broadcast. The number(s) in parentheses next to each node are the labels of nodes from which data has been received prior to the current communication step. Only the first, second, and last communication steps are shown.


1 (6)
1 (5)
1 (4)
7
6
5
4
(7)
(6)
(5)
(4)
1 (7)
1 (3)
(0)
(1)
(2)
(3)
0
1
2
3
1 (0)
1 (1)
1 (2)
1st communication step
2 (5)
2 (4)
2 (3)
(7,6)
(6,5)
(5,4)
(4,3)
2 (6)
2 (2)
(0,7)
(1,0)
(2,1)
(3,2)
2 (7)
2 (0)
2 (1)
2nd communication step
7 (0)
7 (7)
7 (6)
(7,6,5,4,3,2,1)
(6,5,4,3,2,1,0)
(5,4,3,2,1,0,7)
(4,3,2,1,0,7,6)
7 (1)
7 (5)
(0,7,6,5,4,3,2)
(1,0,7,6,5,4,3)
(2,1,0,7,6,5,4)
(3,2,1,0,7,6,5)
7 (2)
7 (3)
7 (4)
7th communication step

3.3 All-Reduce and Prefix-Sum Operations

The communication pattern of all-to-all broadcast can be used to perform some other operations as well. One of these operations is a third variation of reduction, in which each node starts with a buffer of size m and the final results of the operation are identical buffers of size m on each node that are formed by combining the original p buffers using an associative operator.

Semantically, this operation, often referred to as the all-reduce operation, is identical to performing an all-to-one reduction followed by a one-to-all broadcast of the result.

This operation is different from all-to-all reduction, in which p simultaneous all-to-one reductions take place, each with a different destination for the result.

An all-reduce operation with a single-word message on each node is often used to implement barrier synchronization on a message-passing computer.

The semantics of the reduction operation are such that, while executing a parallel program, no node can finish the reduction before each node has contributed a value.

A simple method to perform all-reduce is to perform an all-to-one reduction followed by a one-to-all broadcast.

However, there is a faster way to perform all-reduce by using the communication pattern of all-to-all broadcast. Figure 3.6. illustrates this algorithm for an eight-node hypercube.

Assume that each integer in parentheses in the figure, instead of denoting a message, denotes a number to be added that originally resided at the node with that integer label.

To perform reduction, we follow the communication steps of the all-to-all broadcast procedure, but at the end of each step, add two numbers instead of concatenating two messages.

At the termination of the reduction procedure, each node holds the sum $(0 + 1 + 2 + \cdots + 7)$ (rather than eight messages numbered from 0 to 7, as in the case of all-to-all broadcast).

Unlike all-to-all broadcast, each message transferred in the reduction operation has only one word. The size of the messages does not double in each step because the numbers are added instead of being concatenated. Therefore, the total communication time for all log p steps is

Equation 4.5

$$T = (t_s + t_w m) \log p.$$

Finding prefix sums (also known as the scan operation) is another important problem that can be solved by using a communication pattern similar to that used in all-to-all broadcast and all-reduce operations. Given p numbers n_0, n_1, ..., n_{p-1} (one on each node), the problem is to compute

the sums $s_k = \Sigma_{i=0}^{k} n_i$ for all k between 0 and p - 1. For example, if the original sequence of numbers is <3, 1, 4, 0, 2>, then the sequence of prefix sums is <3, 4, 8, 8, 10>.

Initially, n_k resides on the node labeled k, and at the end of the procedure, the same node holds s_k . Instead of starting with a single numbers, each node could start with a buffer or vector of size m and the m-word result would be the sum of the corresponding elements of buffers.

Figure 3.13 illustrates the prefix sums procedure for an eight-node hypercube. This figure is a modification of Figure 3.11.

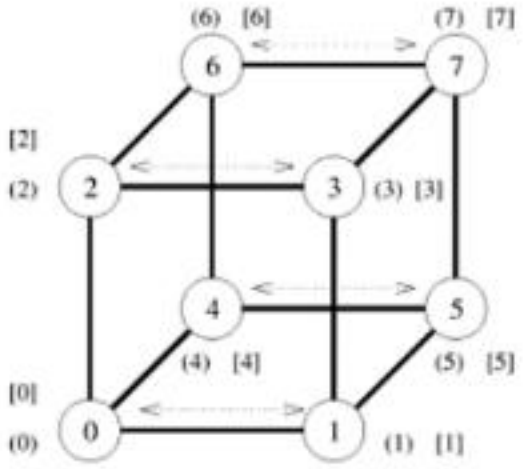
The modification is required to accommodate the fact that in prefix sums the node with label k uses information from only the k-node subset of those nodes whose labels are less than or equal to k. To accumulate the correct prefix sum, every node maintains an additional result buffer.

This buffer is denoted by square brackets in Figure 3.13. At the end of a communication step, the content of an incoming message is added to the result buffer only if the message comes from a node with a smaller label than that of the recipient node.

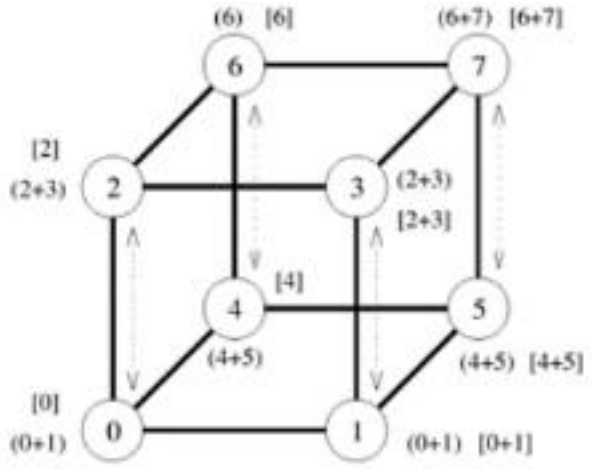
The contents of the outgoing message (denoted by parentheses in the figure) are updated with every incoming message, just as in the case of the all-reduce operation.

For instance, after the first communication step, nodes 0, 2, and 4 do not add the data received from nodes 1, 3, and 5 to their result buffers. However, the contents of the outgoing messages for the next step are updated.

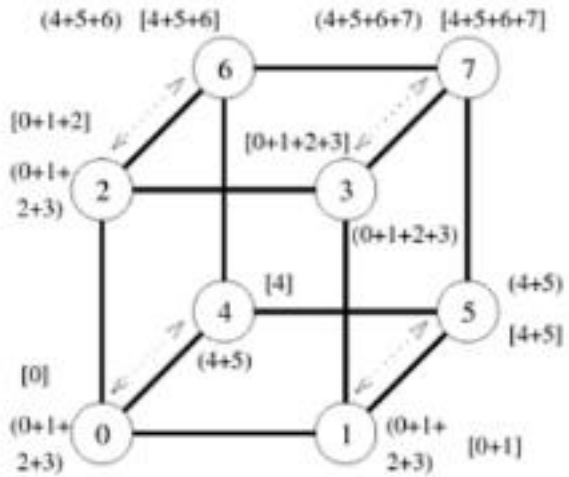
Figure 3.13. Computing prefix sums on an eight-node hypercube. At each node, square brackets show the local prefix sum accumulated in the result buffer and parentheses enclose the contents of the outgoing message buffer for the next step.



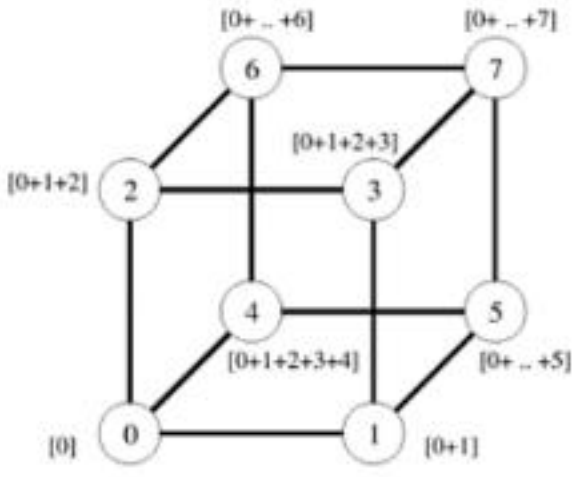
(a) Initial distribution of values



(b) Distribution of sums before second step



(c) Distribution of sums before third step



(d) Final distribution of prefix sums

Since not all of the messages received by a node contribute to its final result, some of the messages it receives may be redundant. We have omitted these steps of the standard all-to-all broadcast communication pattern from Figure 3.13, although the presence or absence of these messages does not affect the results of the algorithm. Algorithm 4.9 gives a procedure to solve the prefix sums problem on a d-dimensional hypercube.

Algorithm 4.9 Prefix sums on a d-dimensional hypercube.

```
1.   procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
2.   begin
3.      result := my_number;
4.      msg := result;
5.      for i := 0 to d - 1 do
6.          partner := my_id XOR 2^i;
7.          send msg to partner;
8.          receive number from partner;
9.          msg := msg + number;
10.         if (partner < my_id) then result := result + number;
11.     endfor;
12.  end PREFIX_SUMS_HCUBE
```

3.4 Scatter and Gather

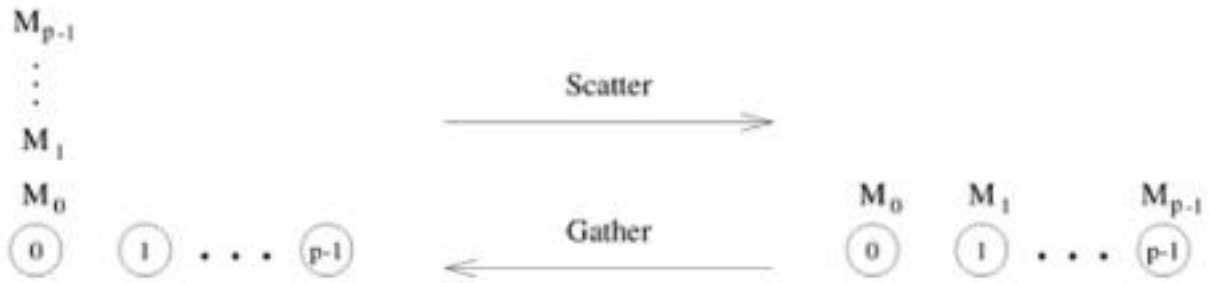
In the scatter operation, a single node sends a unique message of size m to every other node. This operation is also known as one-to-all personalized communication.

One-to-all personalized communication is different from one-to-all broadcast in that the source node starts with p unique messages, one destined for each node.

Unlike one-to-all broadcast, one-to-all personalized communication does not involve any duplication of data. The dual of one-to-all personalized communication or the scatter operation is the gather operation, or concatenation, in which a single node collects a unique message from each node.

A gather operation is different from an all-to-one reduce operation in that it does not involve any combination or reduction of data. Figure 3.14 illustrates the scatter and gather operations.

Figure 3.14. Scatter and gather operations.



Although the scatter operation is semantically different from one-to-all broadcast, the scatter algorithm is quite similar to that of the broadcast.

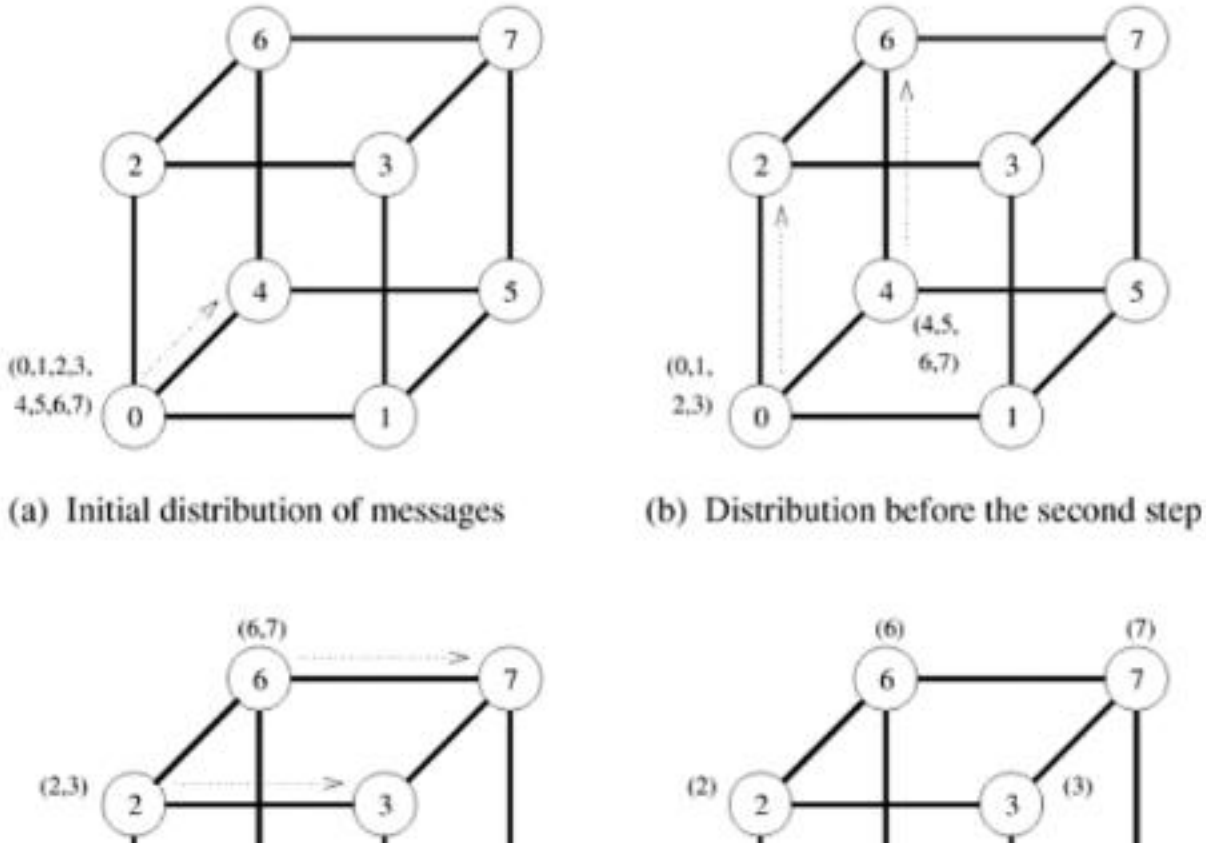
Figure 3.15 shows the communication steps for the scatter operation on an eight-node hypercube. The communication patterns of one-to-all broadcast (Figure 3.6) and scatter (Figure 3.15) are identical.

Only the size and the contents of messages are different. In Figure 3.15, the source node (node 0) contains all the messages.

The messages are identified by the labels of their destination nodes. In the first communication step, the source transfers half of the messages to one of its neighbors.

In subsequent steps, each node that has some data transfers half of it to a neighbor that has yet to receive any data. There is a total of $\log p$ communication steps corresponding to the $\log p$ dimensions of the hypercube.

Figure 3.15. The scatter operation on an eight-node hypercube.



(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages

The gather operation is simply the reverse of scatter. Each node starts with an m word message.

In the first step, every odd numbered node sends its buffer to an even numbered neighbor behind it, which concatenates the received message with its own buffer.

Only the even numbered nodes participate in the next communication step which results in nodes with multiples of four labels gathering more data and doubling the sizes of their data.

The process continues similarly, until node 0 has gathered the entire data.

Just like one-to-all broadcast and all-to-one reduction, the hypercube algorithms for scatter and gather can be applied unaltered to linear array and mesh interconnection topologies without any increase in the communication time.

Cost Analysis All links of a p-node hypercube along a certain dimension join two p/2-node subcubes .

As Figure 3.15 illustrates, in each communication step of the scatter operations, data flow from one subcube to another.

The data that a node owns before starting communication in a certain dimension are such that half of them need to be sent to a node in the other subcube.

In every step, a communicating node keeps half of its data, meant for the nodes in its subcube, and sends the other half to its neighbor in the other subcube.

The time in which all data are distributed to their respective destinations is

Equation 3.6

$$T = t_s \log p + t_w m(p-1).$$

The scatter and gather operations can also be performed on a linear array and on a 2-D square mesh in time $t_s \log p + t_w m(p - 1)$.

Note that disregarding the term due to message-startup time, the cost of scatter and gather operations for large messages on any k-d mesh interconnection network is similar.

In the scatter operation, at least $m(p - 1)$ words of data must be transmitted out of the source node, and in the gather operation, at least $m(p - 1)$ words of data must be received by the destination node.

Therefore, as in the case of all-to-all broadcast, $t_w m(p - 1)$ is a lower bound on the communication time of scatter and gather operations. This lower bound is independent of the interconnection network.

3.5 All-to-All Personalized Communication

In all-to-all personalized communication, each node sends a distinct message of size m to every other node. Each node sends different messages to different nodes, unlike all-to-all broadcast, in which each node sends the same message to all other nodes.

Figure 3.16 illustrates the all-to-all personalized communication operation.

A careful observation of this figure would reveal that this operation is equivalent to transposing a two-dimensional array of data distributed among p processes using one-dimensional array partitioning.

All-to-all personalized communication is also known as total exchange.

This operation is used in a variety of parallel algorithms such as fast Fourier transform, matrix transpose, sample sort, and some parallel database join operations.

Figure 3.16. All-to-all personalized communication.

Example 3.2 Matrix transposition

The transpose of an n x n matrix A is a matrix A^T of the same size, such that $A^T[i, j] = A[j, i]$ for $0 \leq i, j < n$.

Consider an n x n matrix mapped onto n processors such that each processor contains one full row of the matrix.

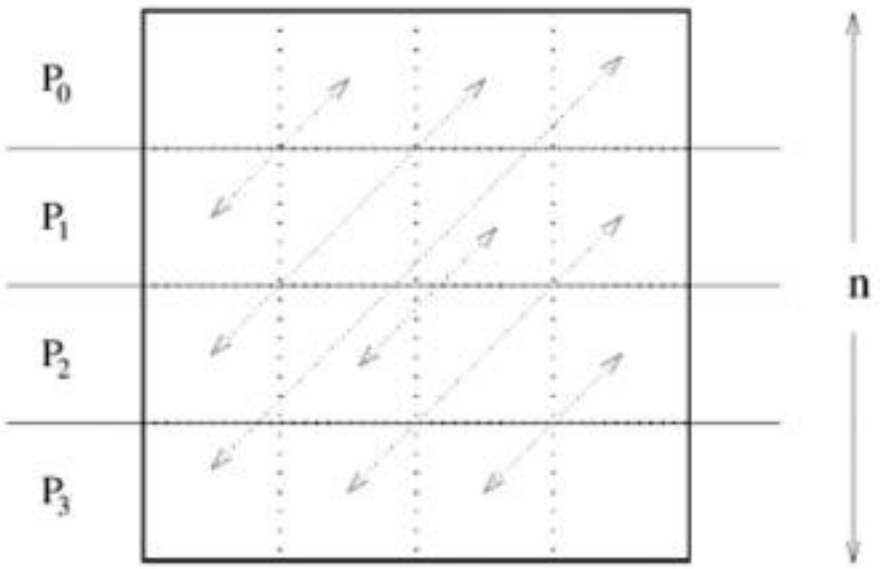
With this mapping, processor P_i initially contains the elements of the matrix with indices [i, 0], [i, 1], ..., [i, n - 1].

After the transposition, element [i, 0] belongs to P_0, element [i, 1] belongs to P_1, and so on. In general, element [i, j] initially resides on P_i, but moves to P_j during the transposition.

The data-communication pattern of this procedure is shown in Figure 3.17 for a 4 x 4 matrix mapped onto four processes using one-dimensional row wise partitioning.

Note that in this figure every processor sends a distinct element of the matrix to every other processor. This is an example of all-to-all personalized communication.

Figure 3.17. All-to-all personalized communication in transposing a 4 x 4 matrix using four processes.



In general, if we use p processes such that p $\leq$ n, then each process initially holds n/p rows (that is, n^2/p elements) of the matrix. Performing the transposition now involves an all-to-all personalized communication of matrix blocks of size n/p x n/p, instead of individual elements.

3.6 Circular Shift

Circular shift is a member of a broader class of global communication operations known as permutation.

A permutation is a simultaneous, one-to-one data redistribution operation in which each node sends a packet of m words to a unique node.

We define a circular q-shift as the operation in which node i sends a data packet to node (i + q) mod p in a p-node ensemble (0 < q < p).

The shift operation finds application in some matrix computations and in string and image pattern matching.

3.6.1 Mesh

The implementation of a circular q-shift is fairly intuitive on a ring or a bidirectional linear array. It can be performed by min{q , p - q} neighbor-to-neighbor communications in one direction.

Mesh algorithms for circular shift can be derived by using the ring algorithm.

If the nodes of the mesh have row-major labels, a circular q-shift can be performed on a p-node square wraparound mesh in two stages.

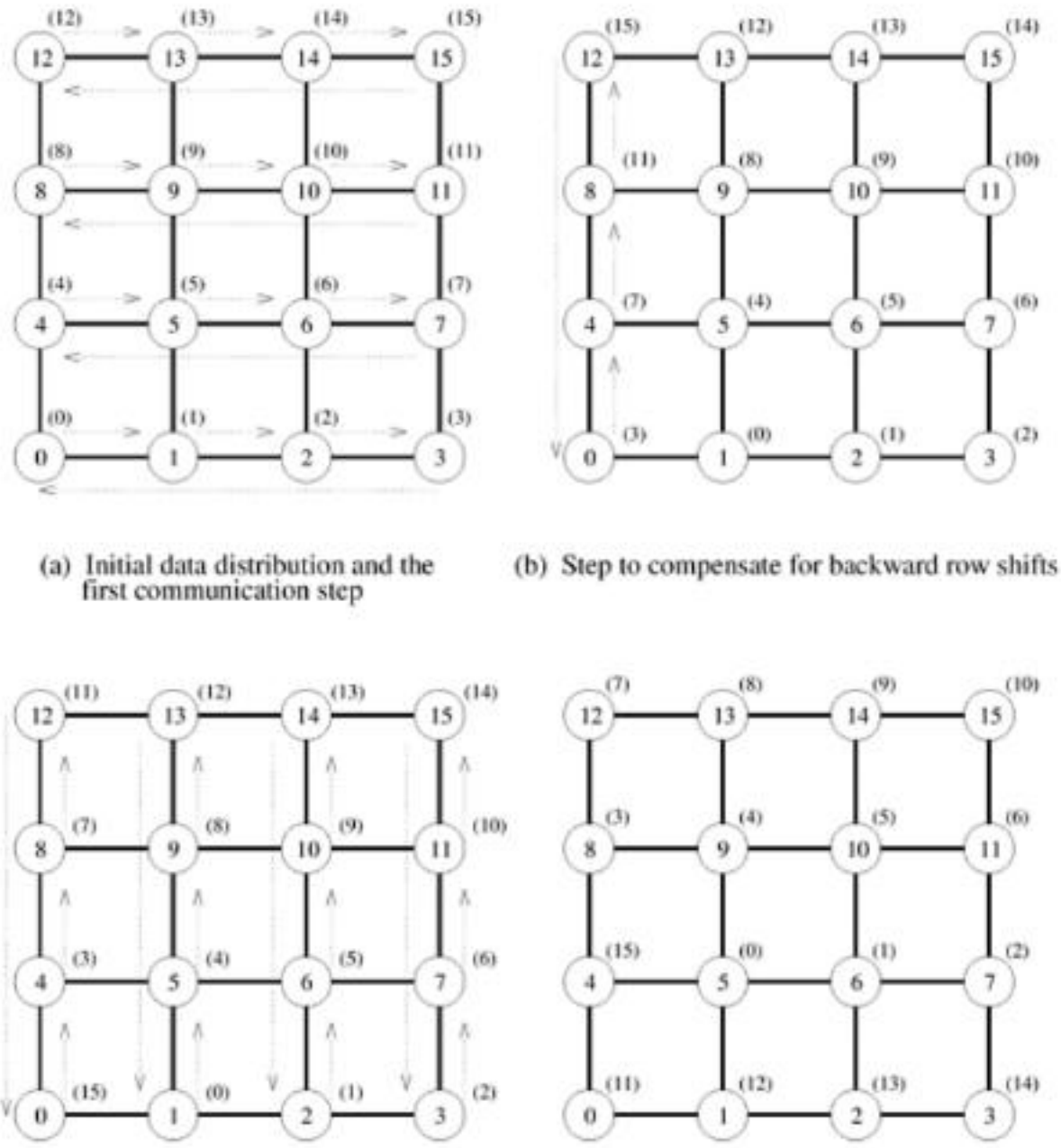
This is illustrated in Figure 3.22 for a circular 5-shift on a 4 x 4 mesh.

First, the entire set of data is shifted simultaneously by (q mod $\sqrt{p}$) steps along the rows.

Then it is shifted by $\lfloor q/\sqrt{p} \rfloor$ steps along the columns. During the circular row shifts, some of the data traverse the wraparound connection from the highest to the lowest labeled nodes of the rows.

All such data packets must shift an additional step forward along the columns to compensate for the $\sqrt{p}$ distance that they lost while traversing the backward edge in their respective rows. For example, the 5-shift in Figure 3.22 requires one row shift, a compensatory column shift, and finally one column shift.

Figure 3.22. The communication steps in a circular 5-shift on a 4 x 4 mesh.



(a) Initial data distribution and the first communication step

(b) Step to compensate for backward row shifts

(c) Column shifts in the third communication step

(d) Final distribution of the data

In practice, we can choose the direction of the shifts in both the rows and the columns to minimize the number of steps in a circular shift. For instance, a 3-shift on a 4 x 4 mesh can be performed by a single backward row shift. Using this strategy, the number of unit shifts in a direction cannot exceed $\lfloor \sqrt{p}/2 \rfloor$

Cost Analysis Taking into account the compensating column shift for some packets, the total time for any circular q-shift on a p-node mesh using packets of size m has an upper bound of

$$T = (t_s + t_w m)(\sqrt{p} + 1)$$

3.6.2 Hypercube

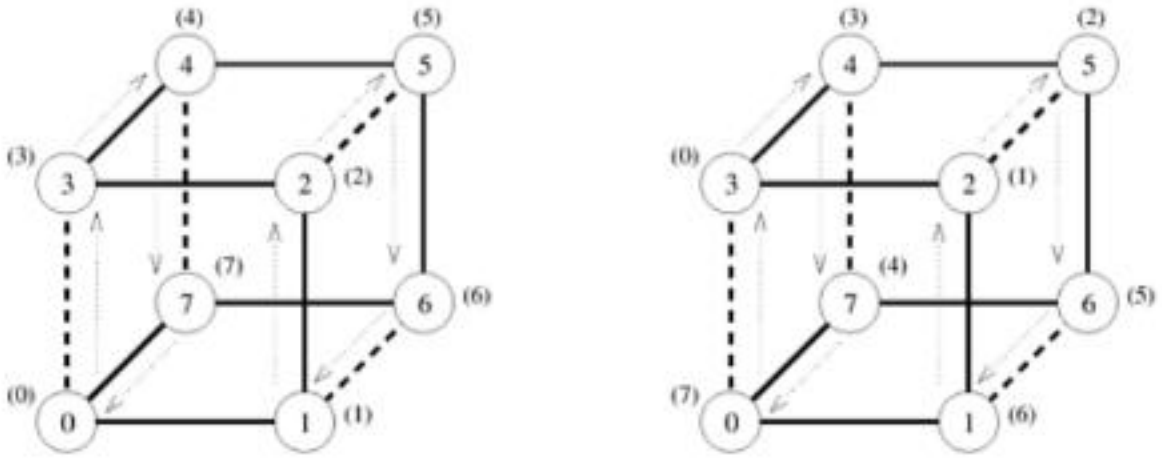
In developing a hypercube algorithm for the shift operation, we map a linear array with 2^d nodes onto a d-dimensional hypercube.

We do this by assigning node i of the linear array to node j of the hypercube such that j is the d-bit binary reflected Gray code (RGC) of i. Figure 3.23 illustrates this mapping for eight nodes.

A property of this mapping is that any two nodes at a distance of 2^i on the linear array are separated by exactly two links on the hypercube.

An exception is i = 0 (that is, directly-connected nodes on the linear array) when only one hypercube link separates the two nodes.

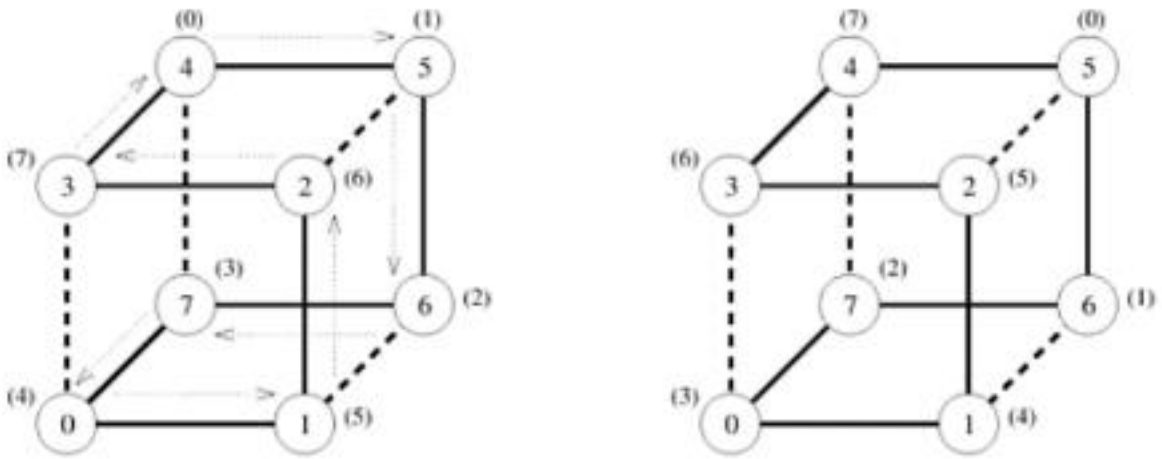
Figure 3.23. The mapping of an eight-node linear array onto a three-dimensional hypercube to perform a circular 5-shift as a combination of a 4-shift and a 1-shift.



First communication step of the 4-shift

Second communication step of the 4-shift

(a) The first phase (a 4-shift)



(b) The second phase (a 1-shift)

(c) Final data distribution after the 5-shift

To perform a q-shift, we expand q as a sum of distinct powers of 2. The number of terms in the sum is the same as the number of ones in the binary representation of q.

For example, the number 5 can be expressed as $2^2 + 2^0$. These two terms correspond to bit positions 0 and 2 in the binary representation of 5, which is 101. If q is the sum of s distinct powers of 2, then the circular q-shift on a hypercube is performed in s phases.

In each phase of communication, all data packets move closer to their respective destinations by short cutting the linear array (mapped onto the hypercube) in leaps of the powers of 2.

For example, as Figure 3.23 shows, a 5-shift is performed by a 4-shift followed by a 1-shift. The number of communication phases in a q-shift is exactly equal to the number of ones in the binary representation of q.

Each phase consists of two communication steps, except the 1-shift, which, if required (that is, if the least significant bit of q is 1), consists of a single step. For example, in a 5-shift, the first phase of a 4-shift (Figure 3.23(a)) consists of two steps and the second phase of a 1-shift (Figure 3.23(b)) consists of one step.

Thus, the total number of steps for any q in a p-node hypercube is at most 2 log p - 1.

All communications in a given time step are congestion-free.

This is ensured by the property of the linear array mapping that all nodes whose mutual distance on the linear array is a power of 2 are arranged in disjoint subarrays on the hypercube.

Thus, all nodes can freely communicate in a circular fashion in their respective subarrays. This is shown in Figure 3.23(a), in which nodes labeled 0, 3, 4, and 7 form one subarray and nodes labeled 1, 2, 5, and 6 form another subarray.

The upper bound on the total communication time for any shift of m-word packets on a p-node hypercube is

Equation 3.11

$$T = (t_s + t_w m)(2 \log p - 1).$$

We can reduce this upper bound to $(t_s + t_w m) \log p$ by performing both forward and backward shifts. For example, on eight nodes, a 6-shift can be performed by a single backward 2-shift instead of a forward 4-shift followed by a forward 2-shift.

We now show that if the E-cube routing introduced in Section 3.5 is used, then the time for circular shift on a hypercube can be improved by almost a factor of log p for large messages. This is because with E-cube routing, each pair of nodes with a constant distance l ($i \leq l < p$) has a congestion-free path (Problem 3.22) in a p-node hypercube with bidirectional channels.

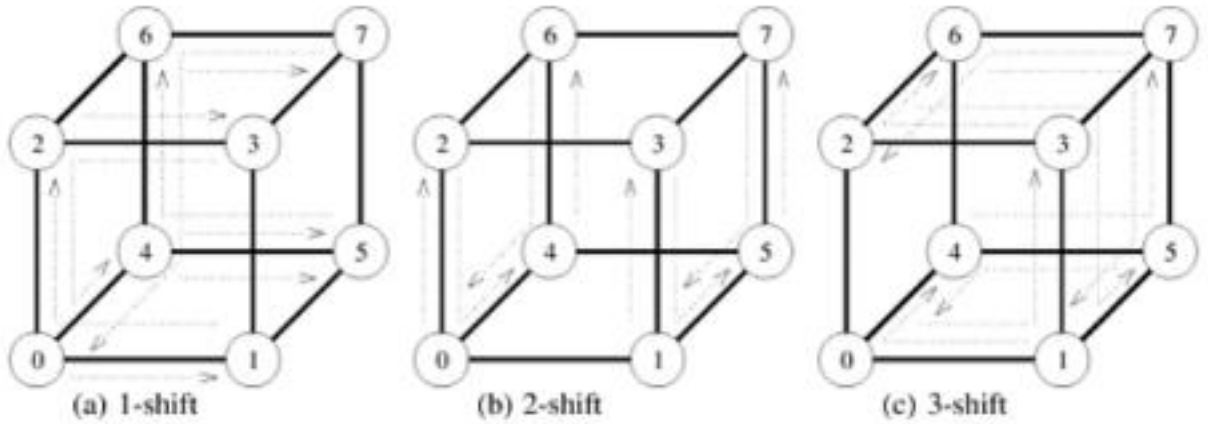
Figure 3.24 illustrates the non-conflicting paths of all the messages in circular q -shift operations for $1 \leq q < 8$ on an eight-node hypercube. In a circular q-shift on a p-node hypercube, the longest path contains log p - $\gamma(q)$ links, where $\gamma(q)$ is the highest integer j such that q is divisible by 2^j (Problem 3.23).

Thus, the total communication time for messages of length m is

Equation 3.12

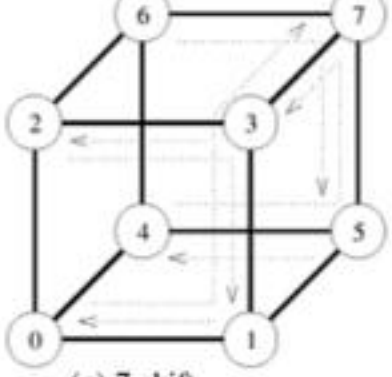
$$T = t_s + t_w m.$$

Figure 3.24. Circular q-shifts on an 8-node hypercube for $1 \leq q < 8$.



(a) 1-shift (b) 2-shift (c) 3-shift

(d) 4-shift (e) 5-shift (f) 6-shift



(g) 7-shift

3.7 Improving the Speed of Some Communication Operations

So far in this chapter, we have derived procedures for various communication operations and their communication times under the assumptions that the original messages could not be split into smaller parts and that each node had a single port for sending and receiving data.

In this section, we briefly discuss the impact of relaxing these assumptions on some of the communication operations.

3.7.1 Splitting and Routing Messages in Parts

In the procedures described in Sections 3.1-3.6, we assumed that an entire m-word packet of data travels between the source and the destination nodes along the same path.

If we split large messages into smaller parts and then route these parts through different paths, we can sometimes utilize the communication network better.

We have already shown that, with a few exceptions like one-to-all broadcast, all-to-one reduction, all-reduce, etc., the communication operations discussed in this chapter are asymptotically optimal for large messages; that is, the terms associated with t_w in the costs of these operations cannot be reduced asymptotically.

In this section, we present asymptotically optimal algorithms for three global communication operations.

Note that the algorithms of this section rely on m being large enough to be split into p roughly equal parts. Therefore, the earlier algorithms are still useful for shorter messages.

A comparison of the cost of the algorithms in this section with those presented earlier in this chapter for the same operations would reveal that the term associated with t_s increases and the term associated with t_w decreases when the messages are split.

Therefore, depending on the actual values of t_s, t_w, and p, there is a cut-off value for the message size m and only the messages longer than the cut-off would benefit from the algorithms in this section.

One-to-All Broadcast

Consider broadcasting a single message M of size m from one source node to all the nodes in a p-node ensemble.

If m is large enough so that M can be split into p parts M_0, M_1, ..., M_{p-1} of size m/p each, then a scatter operation (Section 3.4) can place M_i on node i in time $t_s \log p + t_w(m/p)(p - 1)$.

Note that the desired result of the one-to-all broadcast is to place $M = M_0 \cup M_1 \cup \ldots \cup M_{p-1}$ on all nodes. This can be accomplished by an all-to-all broadcast of the messages of size m/p residing on each node after the scatter operation. This all-to-all broadcast can be completed in time $t_s \log p + t_w(m/p)(p - 1)$ on a hypercube. Thus, on a hypercube, one-to-all broadcast can be performed in time

Equation 3.13

$$\begin{aligned} T &= 2 \times (t_s \log p + t_w(p-1)\frac{m}{p}) \\ &\approx 2 \times (t_s \log p + t_w m). \end{aligned}$$

Compared to Equation 3.1, this algorithm has double the startup cost, but the cost due to the t_w term has been reduced by a factor of (log p)/2.

Similarly, one-to-all broadcast can be improved on linear array and mesh interconnection networks as well.

All-to-One Reduction

All-to-one reduction is a dual of one-to-all broadcast. Therefore, an algorithm for all-to-one reduction can be obtained by reversing the direction and the sequence of communication in one-to-all broadcast.

We showed above how an optimal one-to-all broadcast algorithm can be obtained by performing a scatter operation followed by an all-to-all broadcast.

Therefore, using the notion of duality, we should be able to perform an all-to-one reduction by performing all-to-all reduction (dual of all-to-all broadcast) followed by a gather operation (dual of scatter).

We leave the details of such an algorithm as an exercise for the reader (Problem 4.17).

All-Reduce

Since an all-reduce operation is semantically equivalent to an all-to-one reduction followed by a one-to-all broadcast, the asymptotically optimal algorithms for these two operations presented above can be used to construct a similar algorithm for the all-reduce operation.

Breaking all-to-one reduction and one-to-all broadcast into their component operations, it can be shown that an all-reduce operation can be accomplished by an all-to-all reduction followed by a gather followed by a scatter followed by an all-to-all broadcast.

Since the intermediate gather and scatter would simply nullify each other's effect, all-reduce just requires an all-to-all reduction and an all-to-all broadcast.

First, the m-word messages on each of the p nodes are logically split into p components of size roughly m/p words. Then, an all-to-all reduction combines all the i th components on p_i.

After this step, each node is left with a distinct m/p-word component of the final result. An all-to-all broadcast can construct the concatenation of these components on each node.

A p-node hypercube interconnection network allows all-to-one reduction and one-to-all broadcast involving messages of size m/p in time $t_s \log p + t_w(m/p)(p - 1)$ each. Therefore, the all-reduce operation can be completed in time

Equation 3.14

$$T = 2 \times (t_s \log p + t_w(p-1)\frac{m}{p}) \approx 2 \times (t_s \log p + t_w m).$$

3.7.2 All-Port Communication

In a parallel architecture, a single node may have multiple communication ports with links to other nodes in the ensemble.

For example, each node in a two-dimensional wraparound mesh has four ports, and each node in a d-dimensional hypercube has d ports.

In this book, we generally assume what is known as the single-port communication model. In single-port communication, a node can send data on only one of its ports at a time.

Similarly, a node can receive data on only one port at a time. However, a node can send and receive data simultaneously, either on the same port or on separate ports.

In contrast to the single-port model, an all-port communication model permits simultaneous communication on all the channels connected to a node.

On a p-node hypercube with all-port communication, the coefficients of t_w in the expressions for the communication times of one-to-all and all-to-all broadcast and personalized communication are all smaller than their single-port counterparts by a factor of log p.

Since the number of channels per node for a linear array or a mesh is constant, all-port communication does not provide any asymptotic improvement in communication time on these architectures.

Despite the apparent speedup, the all-port communication model has certain limitations.

For instance, not only is it difficult to program, but it requires that the messages are large enough to be split efficiently among different channels. In several parallel algorithms, an increase in the size of messages means a corresponding increase in the granularity of computation at the nodes.

When the nodes are working with large data sets, the internode communication time is dominated by the computation time if the computational complexity of the algorithm is higher than the communication complexity.

For example, in the case of matrix multiplication, there are n^3 computations for n^2 words of data transferred among the nodes.

If the communication time is a small fraction of the total parallel run time, then improving the communication by using sophisticated techniques is not very advantageous in terms of the overall run time of the parallel algorithm.

Another limitation of all-port communication is that it can be effective only if data can be fetched and stored in memory at a rate sufficient to sustain all the parallel communication.

For example, to utilize all-port communication effectively on a p-node hypercube, the memory bandwidth must be greater than the communication bandwidth of a single channel by a factor of at least log p; that is, the memory bandwidth must increase with the number of nodes to support simultaneous communication on all ports.

Some modern parallel computers, like the IBM SP, have a very natural solution for this problem. Each node of the distributed-memory parallel computer is a NUMA shared-memory multiprocessor.

Multiple ports are then served by separate memory banks and full memory and communication bandwidth can be utilized if the buffers for sending and receiving data are placed appropriately across different memory banks.

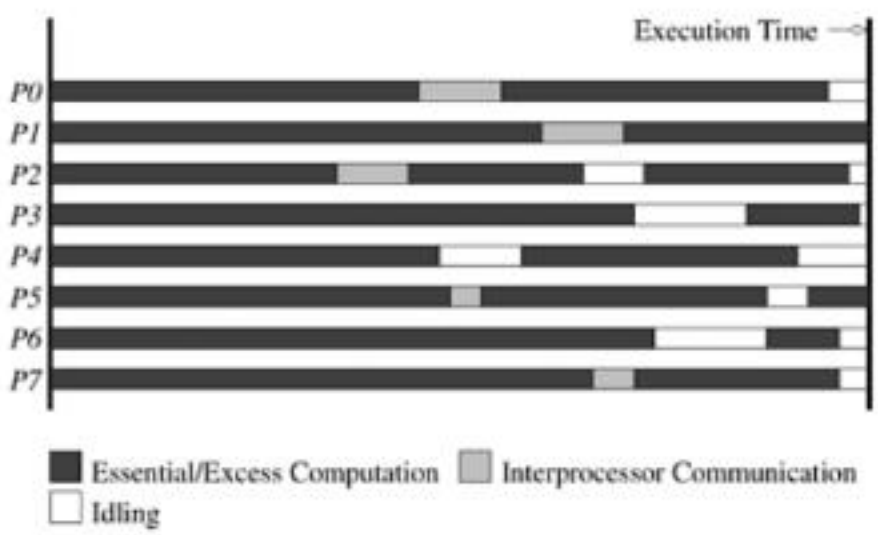
UNIT – IV Analytical Modeling of Parallel Programs

4.1 Sources of Overhead in Parallel Programs

Using twice as many hardware resources, one can rationally expect a program to run twice as fast. However, in typical parallel programs, this is not often the case, due to a variety of outlay associated with parallelism. An accurate quantification of these outlay are critical to the understanding of parallel program performance.

A typical execution outline of a parallel program is illustrated in Figure 4.1. In addition to performing important computation (i.e., computation that would be performed by the serial program for solving the same problem case), a parallel program may also spend time in interprocess communication, idling, and surplus computation (computation not performed by the serial formulation).

Figure 4.1. The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.



Interprocess Interaction Any nontrivial parallel system requires its processing elements to cooperate and communicate data (e.g., intermediate results). The time spent communicating data between processing elements is usually the most important source of parallel processing overhead.

Idling Processing elements in a parallel system may become inactive due to many reasons such as load imbalance, synchronization, and presence of serial components in a program. In many parallel applications (for example, when task generation is dynamic), it is impossible (or at least difficult) to forecast the size of the subtasks assigned to various processing elements. Hence, the problem cannot be subdivided statically amid the processing elements while maintaining uniform

workload. If different processing elements have different workloads, some processing elements may be idle during part of the time that others are working on the problem. In some parallel programs, processing elements must synchronize at firm points during parallel program execution. If all processing elements are not ready for synchronization at the same time, then the ones that are ready earlier will be idle until all the rest are ready. Parts of an algorithm may be unparallelizable, allowing only a single processing element to work on it. While one processing element works on the serial part, all the other processing elements must wait.

Excess Computation The best ever known sequential algorithm for a problem may be difficult or impossible to parallelize, forcing us to use a parallel algorithm based on a poorer but easily parallelizable (that is, one with a higher degree of concurrency) sequential algorithm. The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

A parallel algorithm based on the best serial algorithm may still perform more combined computation than the serial algorithm. An example of such a computation is the Fast Fourier Transform algorithm. In its serial version, the results of certain computations can be reused. However, in the parallel version, these results cannot be reused because they are generated by different processing elements. Therefore, some computations are performed numerous times on different processing elements.

Since different parallel algorithms for solving the same problem acquire varying overheads, it is important to quantify these overheads with a view to establishing a figure of merit for each algorithm.

4.2 Performance Metrics for Parallel Systems

It is important to study the performance of parallel programs with a view to determining the finest algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of routine analysis.

4.2.1 Execution Time

The serial runtime of a program is the time gone between the beginning and the end of its execution on a sequential computer. The ***parallel runtime*** is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by T_S and the parallel runtime by T_P.

4.2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a sole expression referred to as the ***overhead function***. We define overhead function or ***total overhead*** of a parallel system as the total time together used up by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol T_o.

The total time spent in solving a problem summed above all processing elements is pT_P. T_S units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (T_o) is given by

Equation 4.1

$$T_o = pT_P - T_S.$$

When evaluating a parallel system, we are repeatedly interested in knowing how much performance gain is achieved by parallelizing a given application over a in order implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is distinct as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements. We signify speedup by the symbol S.

Example 4.1 Adding n numbers using n processing elements

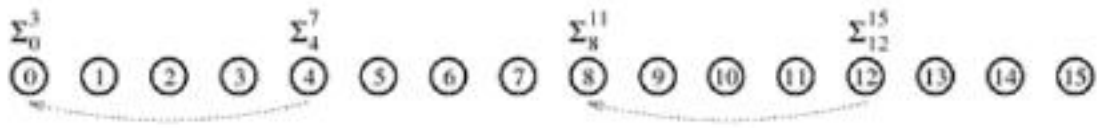
Consider the problem of adding n numbers by using n processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that n is a power of two, we can perform this operation in log n steps by propagating part sums up a logical binary tree of processing elements. Figure 4.2 illustrates the procedure for

$n = 16$. The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from i to j is denoted by Σ_i^j

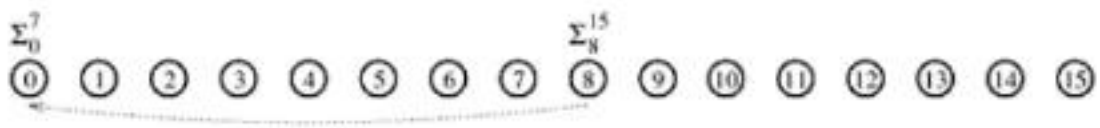
Figure 4.2. Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

(a) Initial data distribution and the first communication step

(b) Second communication step



(c) Third communication step



(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communicatior

Each step shown in Figure 4.2 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say t_c, and the communication of a single word can be performed in time $t_s + t_w$. Therefore, the addition and communication operations take a constant amount of time. Thus,

Equation 4.2

$$T_P = \Theta(\log n).$$

Since the problem can be solved in $Q(n)$ time on a single processing element, its speedup is

Equation 4.3

$$S = \Theta\left(\frac{n}{\log n}\right).$$

For a known problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the slightest amount of time. Given a parallel algorithm, it is fair-haired to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processing element. Sometimes, the asymptotically fastest sequential algorithm to answer a problem is not known, or its runtime has a big constant that makes it impractical to implement. In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the finest sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the similar problem. We formally define the ***speedup*** S as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the similar problem on p processing elements. The p processing elements used by the parallel algorithm are tacit to be identical to the one used by the sequential algorithm.

Example 4.2 Computing speedups of parallel programs

Consider the example of parallelizing bubble sort. Assume that a serial version of bubble sort of 10^5 records takes 150 seconds and a serial quick sort can sort the same list in 30 seconds. If a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on four processing elements, it would appear that the parallel odd-even sort algorithm results in a speedup of 150/40 or 3.75. However, this conclusion is deceptive, as in reality the parallel algorithm results in a speedup of 30/40 or 0.75 with respect to the best serial algorithm.

Theoretically, speedup can not at all exceed the number of processing elements, p. If the finest sequential algorithm takes T_S units of time to solve a given problem on a single processing element, then a speedup of p can be obtained on p processing elements if none of the processing elements spends more than time T_S/p. A speedup greater than p is possible only if each processing element spends less than time T_S/p solving the problem. In this case, a single processing element could emulate the p processing elements and solve the problem in fewer than T_S units of time. This is a disagreement because speedup, by definition, is computed with respect to the best sequential algorithm. If T_S is the serial runtime of the algorithm, then the problem cannot be solved in less than time T_S on a single processing element.

In practice ,a speedup greater than p is occasionally observed (a phenomenon known as ***super linear speedup***). This usually happens when the work performed by a serial algorithm is greater than its parallel formulation or due to hardware features that put the serial implementation at a disadvantage. For example, the data for a problem might be too big to fit into the cache of a single processing element, thereby degrading its performance due to the use of slower memory elements. But when partitioned among several processing elements, the individual data-partitions would be small enough to fit into their respective processing elements' caches. In the remainder of this book, we disregard super linear speedup due to hierarchical memory.

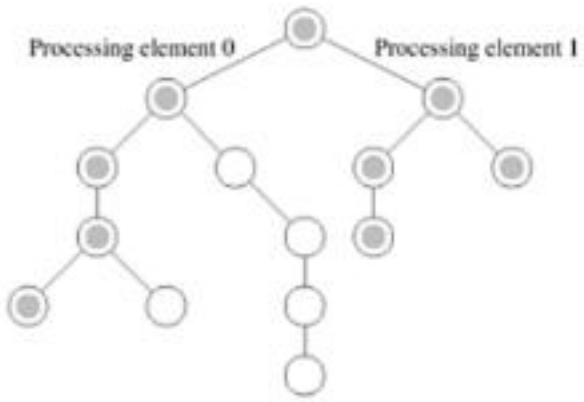
Example 4.3 Super linearity effects from caches

Consider the execution of a parallel program on a two-processor parallel system. The program attempts to solve a problem case of size W. With this size and available cache of 64 KB on one processor, the program has a cache hit rate of 80%. Assuming the latency to cache of 2 ns and latency to DRAM of 100 ns, the effective memory access time is 2 x 0.8 + 100 x 0.2, or 21.6 ns. If the computation is memory leap and performs one FLOP/memory access, this corresponds to a processing rate of 46.3 MFLOPS. Now consider a situation when each of the two processors is effectively executing partly of the problem instance (i.e., size $W/2$). At this problem size, the cache hit ratio is expected to be higher, since the effective problem size is smaller. Let us assume that the cache hit ratio is 90%, 8% of the remaining data comes from local DRAM, and the other 2% comes from the remote DRAM (communication overhead). Assuming that remote data access takes 400 ns, this corresponds to an overall access time of 2 x 0.9 + 100 x 0.08 + 400 x 0.02, or 17.8 ns. The corresponding execution rate at each processor is therefore 56.18, for a total execution rate of 112.36 MFLOPS. The speedup in this case is given by the increase in speed over serial formulation, i.e., 112.36/46.3 or 2.43! Here, because of increased cache hit ratio resulting from lower problem size per processor, we notice super linear speedup.

Example 4.4 Super linearity effects due to exploratory decomposition

Consider an algorithm for exploring leaf nodes of an unstructured tree. Each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'. Such computations are often used to solve combinatorial problems, where the label 'S' could imply the solution to the problem. In Figure 4.3, we illustrate such a tree. The solution node is the rightmost leaf in the tree. A serial formulation of this problem based on depth-first tree traversal explores the entire tree, i.e., all 14 nodes. If it takes time t_c to visit a node, the time for this traversal is $14t_c$. Now consider a parallel formulation in which the left sub tree is explored by processing element 0 and the right sub tree by processing element 1. If both processing elements explore the tree at the same speed, the parallel formulation explores only the shaded nodes before the solution is found. Notice that the total work done by the parallel algorithm is only nine node expansions, i.e., $9t_c$. The corresponding parallel time, assuming the root node expansion is serial, is $5t_c$ (one root node expansion, followed by four node expansions by each processing element). The speedup of this two-processor execution is therefore $14t_c/5t_c$, or 2.8!

Figure 4.3. Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left sub tree and processor 1 searching the right sub tree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.



The cause for this super linearity is that the work performed by parallel and serial algorithms is different. Indeed, if the two-processor algorithm was implemented as two processes on the same processing element, the algorithmic super linearity would disappear for this problem instance. Note that when exploratory decomposition is used, the relative amount of work performed by serial and parallel algorithms is dependent upon the location of the solution, and it is often not possible to find a serial algorithm that is optimal for all instances.

4.2.4 Efficiency

Only an ideal parallel system containing p processing elements can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw in Example 4.1, part of the time required by the processing elements to compute the sum of n numbers is spent idling (and communicating in real systems). ***Efficiency*** is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

Equation 4.4

$$E = \frac{S}{p}.$$

Example 4.5 Efficiency of adding n numbers on n processing elements

From Equation 4.3 and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

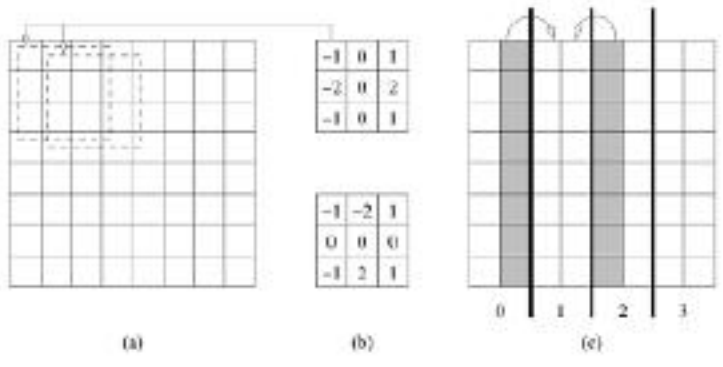
$$\begin{aligned} E &= \frac{\Theta\left(\frac{n}{\log n}\right)}{n} \\ &= \Theta\left(\frac{1}{\log n}\right) \end{aligned}$$

We also illustrate the process of deriving the parallel runtime, speedup, and efficiency while preserving various constants associated with the parallel platform.

Example 4.6 Edge detection on images

Given an n x n pixel image, the problem of detecting edges corresponds to applying a3x 3 template to each pixel. The process of applying the template corresponds to multiplying pixel values with corresponding template values and summing across the template (a convolution operation). This process is illustrated in Figure 4.4(a) along with typical templates (Figure 4.4(b)). Since we have nine multiply-add operations for each pixel, if each multiply-add takes time t_c, the entire operation takes time $9t_cn^2$ on a serial computer.

Figure 4.4. Example of edge detection: (a) an 8 x 8 image; (b) typical templates for detecting edges; and (c) partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1.



A simple parallel algorithm for this problem partitions the image equally across the processing elements and each processing element applies the template to its own sub image. Note that for applying the template to the boundary pixels, a processing element must get data that is assigned to the adjoining processing element. Specifically, if a processing element is assigned a vertically sliced sub image of dimension n x (n/p), it must access a single layer of n pixels from the processing element to the left and a single layer of n pixels from the processing element to the right (note that one of these accesses is redundant for the two processing elements assigned the sub images at the extremities). This is illustrated in Figure 4.4(c).

On a message passing machine, the algorithm executes in two steps: (i) exchange a layer of n pixels with each of the two adjoining processing elements; and (ii) apply template on local sub image. The first step involves two n-word messages (assuming each pixel takes a word to communicate RGB data). This takes time $2(t_s + t_wn)$. The second step takes time $9t_cn^2/p$. The total time for the algorithm is therefore given by:

$$T_P = 9t_c\frac{n^2}{p} + 2(t_s + t_wn)$$

The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

And

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

4.2.5 Cost

We define the ***cost*** of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processing elements.

The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be ***cost-optimal*** if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Q terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of Q(1).

Cost is sometimes referred to as ***work*** or ***processor-time product***, and a cost-optimal system is also known as a pT_p -optimal system.

4.3 The Effect of Granularity on Performance

Example 4.7 illustrated an instance of an algorithm that is not cost-optimal. The algorithm discussed in this example uses as many processing elements as the number of inputs, which is excessive in terms of the number of processing elements. In practice, we assign larger pieces of input data to processing elements. This corresponds to increasing the granularity of computation on the processing elements. Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called ***scaling down*** a parallel system in terms of the number of processing elements. A naive way to scale down a parallel system is to design a parallel algorithm for one input element per processing element, and then use fewer processing elements to simulate a large number of processing elements. If there are n inputs and only p processing elements ($p < n$), we can use the parallel algorithm designed for n processing elements by assuming n virtual processing elements and having each of the p physical processing elements simulate n/p virtual processing elements.

As the number of processing elements decreases by a factor of n/p, the computation at each processing element increases by a factor of n/p because each processing element now performs the work of n/p processing elements. If virtual processing elements are mapped appropriately onto physical processing elements, the overall communication time does not grow by more than

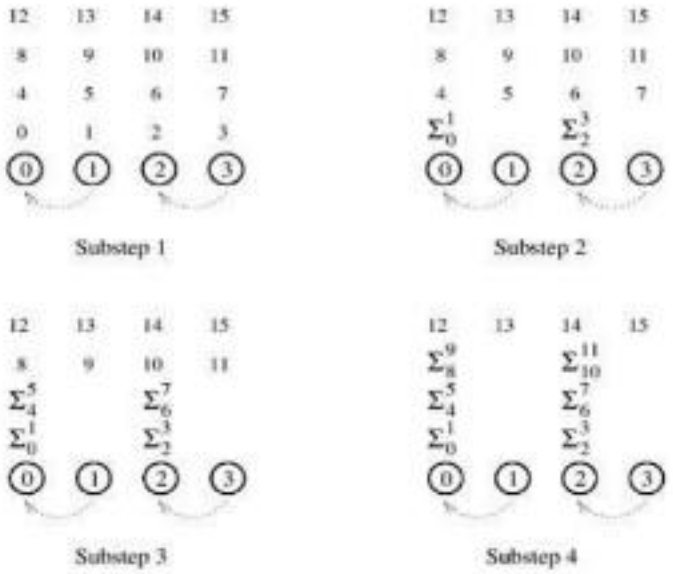
a factor of n/p. The total parallel runtime increases, at most, by a factor of n/p, and the processor-time product does not increase. Therefore, if a parallel system with n processing elements is cost-optimal, using p processing elements (where $p < n$)to simulate n processing elements preserves cost-optimality.

A drawback of this naive method of increasing computational granularity is that if a parallel system is not cost-optimal to begin with, it may still not be cost-optimal after the granularity of computation increases. This is illustrated by the following example for the problem of adding n numbers.

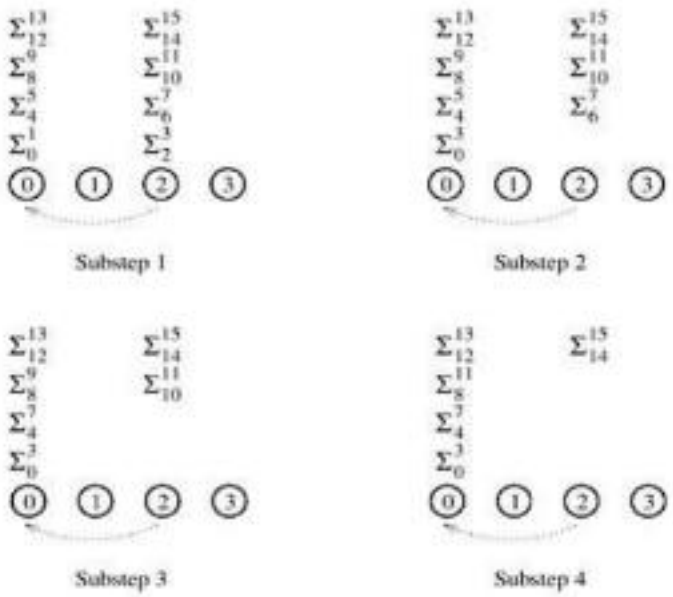
Example 4.9 Adding n numbers on p processing elements

Consider the problem of adding n numbers on p processing elements such that $p < n$ and both n and p are powers of 2. We use the same algorithm as in Example 4.1 and simulate n processing elements on p processing elements. The steps leading to the solution are shown in Figure 4.5 for $n = 16$ and $p = 4$. Virtual processing element i is simulated by the physical processing element labeled $i \bmod p$; the numbers to be added are distributed similarly. The first $\log p$ of the $\log n$ steps of the original algorithm are simulated in $(n/p) \log p$ steps on p processing elements. In the remaining steps, no communication is required because the processing elements that communicate in the original algorithm are simulated by the same processing element; hence, the remaining numbers are added locally. The algorithm takes $\Theta((n/p) \log p)$ time in the steps that require communication, after which a single processing element is left with n/p numbers to add, taking time $\Theta(n/p)$. Thus, the overall parallel execution time of this parallel system is $\Theta((n/p) \log p)$. Consequently, its cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding n numbers sequentially. Therefore, the parallel system is not cost-optimal.

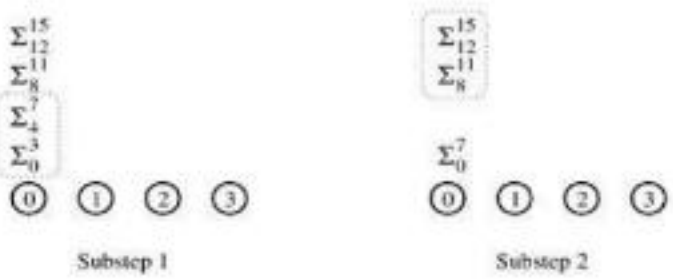
Figure 4.5. Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j. Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).



(a) Four processors simulating the first communication step of 16 processors



(b) Four processors simulating the second communication step of 16 processors



(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step

(e) Final result

Example 4.1 showed that n numbers can be added on an n-processor machine in time $\Theta(\log n)$. When using p processing elements to simulate n virtual processing elements ($p < n$), the expected parallel runtime is $\Theta((n/p)\log n)$. However, in Example 4.9 this task was performed in time $\Theta((n/p)\log p)$ instead. The reason is that every communication step of the original algorithm does not have to be simulated; at times, communication takes place between virtual processing elements that are simulated by the same physical processing element. For these operations, there is no associated overhead. For example, the simulation of the third and fourth steps did not require any communication. However, this reduction in communication was not enough to make the algorithm cost-optimal. Example 4.10 illustrates that the same problem (adding n numbers on p processing elements) can be performed cost-optimally with a smarter assignment of data to processing elements.

Example 4.10 Adding n numbers cost-optimally

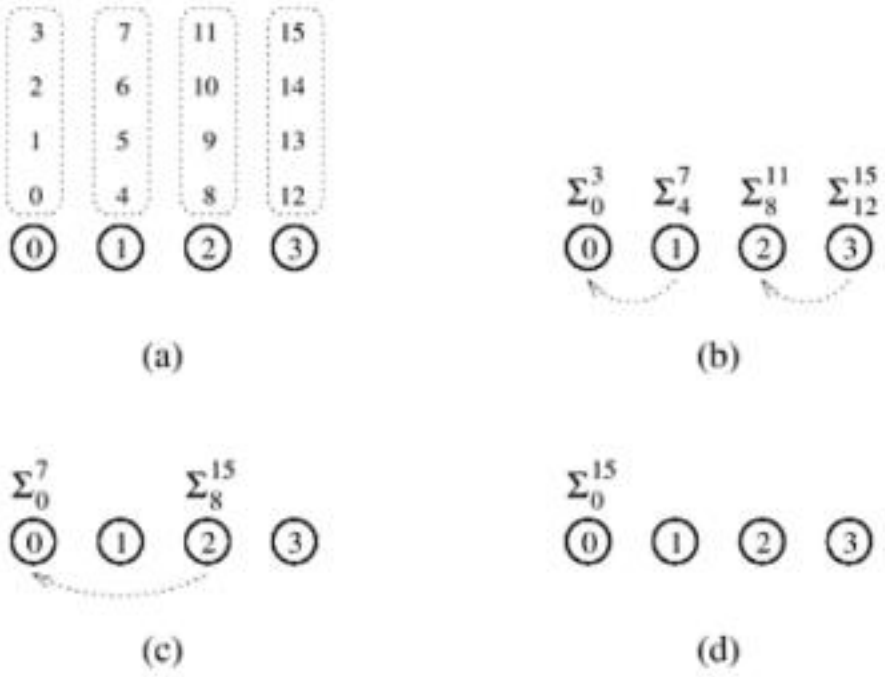
An alternate method for adding n numbers using p processing elements is illustrated in Figure 4.6 for $n = 16$ and $p = 4$. In the first step of this algorithm, each processing element locally adds its n/p numbers in time $\Theta(n/p)$. Now the problem is reduced to adding the p partial sums on p processing elements, which can be done in time $\Theta(\log p)$ by the method described in Example 4.1. The parallel runtime of this algorithm is

Equation 5.5

$$T_P = \Theta(n/p + \log p),$$

and its cost is $\Theta(n + p\log p)$. As long as $n = \Omega(p\log p)$, the cost is $\Theta(n)$, which is the same as the serial runtime. Hence, this parallel system is cost-optimal.

Figure 4.6. A cost-optimal way of computing the sum of 16 numbers using four processing elements.



These simple examples demonstrate that the manner in which the computation is mapped onto processing elements may determine whether a parallel system is cost-optimal. Note, however, that we cannot make all non-cost-optimal systems cost-optimal by scaling down the number of processing elements.

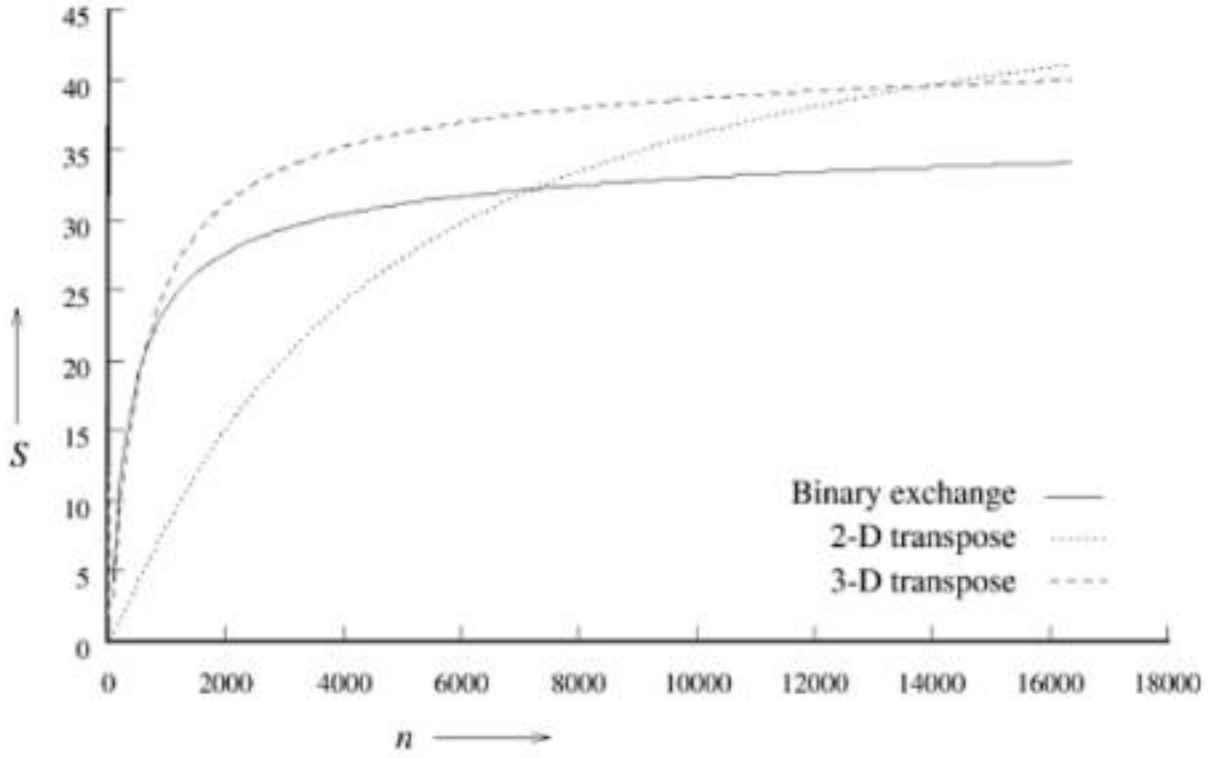
4.4 Scalability of Parallel Systems

Very often, programs are designed and tested for smaller problems on fewer processing elements. However, the real problems these programs are intended to solve are much larger, and the machines contain larger number of processing elements. Whereas code development is simplified by using scaled-down versions of the machine and the problem, their performance and correctness (of programs) is much more difficult to establish based on scaled-down systems. In this section, we will investigate techniques for evaluating the scalability of parallel programs using analytical tools.

Example 4.11 Why is performance extrapolation so difficult?

Consider three parallel algorithms for computing an n-point Fast Fourier Transform (FFT) on 64 processing elements. Figure 5.7 illustrates speedup as the value of n is increased to 18 K. Keeping the number of processing elements constant, at smaller values of n, one would infer from observed speedups that binary exchange and 3-D transpose algorithms are the best. However, as the problem is scaled up to 18 K points or more, it is evident from Figure 5.7 that the 2-D transpose algorithm yields best speedup. (These algorithms are discussed in greater detail in Chapter 13.)

Figure 4.7. A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$ (see Chapter 13 for details).



Similar results can be shown relang to the variaon in number of processing elements as the problem size is held constant. Unfortunately, such parallel performance traces are the norm as opposed to the excepon, making performance predicon based on limited observed data very dicult.

4.4.1 Scaling Characteristics of Parallel Programs

The efficiency of a parallel program can be written as:

$$E = \frac{S}{p} = \frac{T_S}{pT_P}$$

Using the expression for parallel overhead (Equation 4.1), we can rewrite this expression as

Equation 4.6

$$E = \frac{1}{1 + \frac{T_o}{T_S}}.$$

The total overhead function T_o is an increasing function of p. This is because every program must contain some serial component. If this serial component of the program takes time t_{serial} then during this time all the other processing elements must be idle. This corresponds to a total overhead function of $(p - 1) \times t_{serial}$. Therefore, the total overhead function T_o grows at least linearly with p. In addition, due to communication, idling, and excess computation, this function

may grow super linearly in the number of processing elements. Equation 4.6 gives us several interesting insights into the scaling of parallel programs. First, for a given problem size (i.e. the value of T_S remains constant), as we increase the number of processing elements, T_o increases. In such a scenario, it is clear from Equation 4.6 that the overall efficiency of the parallel program goes down. This characteristic of decreasing efficiency with increasing number of processing elements for a given problem size is common to all parallel programs.

Example 4.12 Speedup and efficiency as functions of the number of processing elements

Consider the problem of adding n numbers on p processing elements. We use the same algorithm as in Example 4.10. However, to illustrate actual speedups, we work with constants here instead of asymptotics. Assuming unit time for adding two numbers, the first phase (local summations) of the algorithm takes roughly n/p time. The second phase involves $\log p$ steps with a communication and an addition at each step. If a single communication takes unit time as well, the time for this phase is $2 \log p$. Therefore, we can derive parallel time, speedup, and efficiency as:

Equation 4.7

$$T_P = \frac{n}{p} + 2 \log p$$

Equation 4.8

$$S = \frac{n}{\frac{n}{p} + 2 \log p}$$

Equation 4.9

$$E = \frac{1}{1 + \frac{2p \log p}{n}}$$

These expressions can be used to calculate the speedup and efficiency for any pair of n and p. Figure 4.8 shows the S versus p curves for a few different values of n and p. Table 4.1 shows the corresponding efficiencies.

Figure 4.8. Speedup versus the number of processing elements for adding a list of numbers.

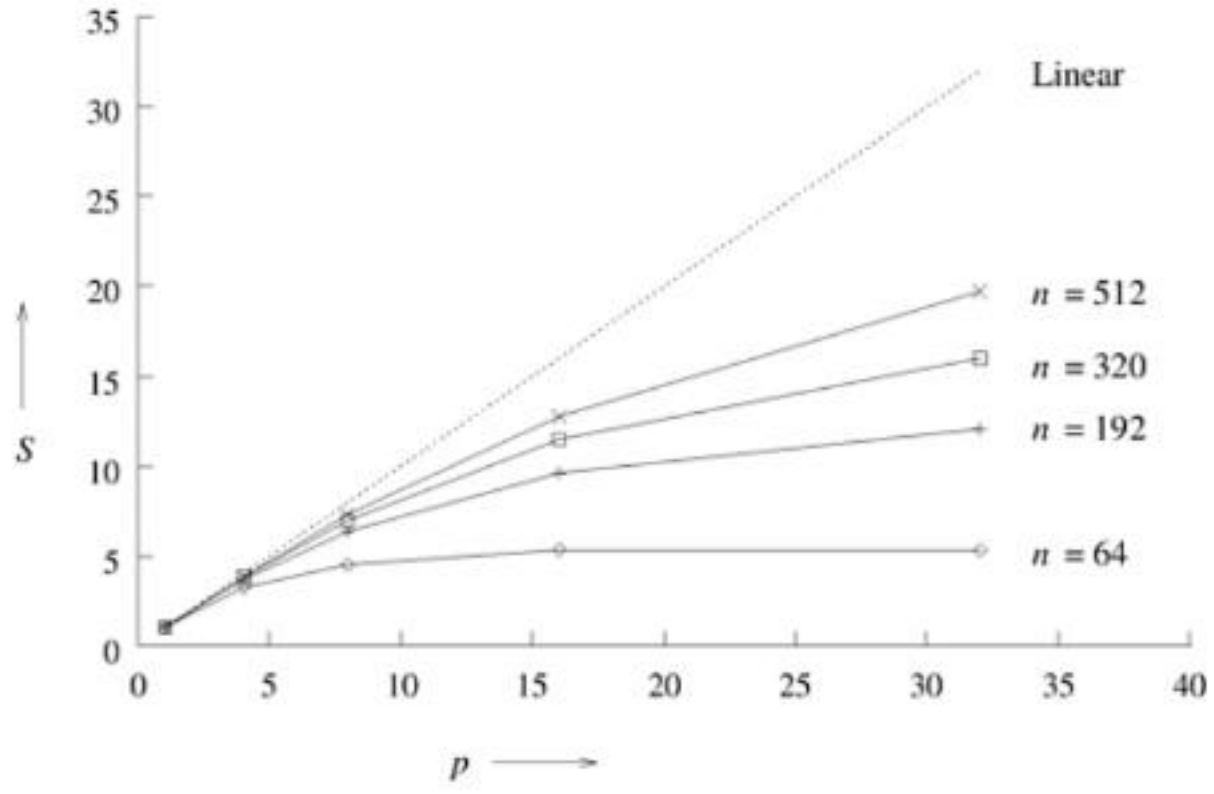


Figure 4.8 and Table 4.1 illustrate that the speedup tends to saturate and efficiency drops as a consequence of ***Amdahl's law***. Furthermore, a larger instance of the same problem yields higher speedup and efficiency for the same number of processing elements, although both speedup and efficiency continue to drop with increasing p.

Let us investigate the effect of increasing the problem size keeping the number of processing elements constant. We know that the total overhead function T_o is a function of both problem size T_S and the number of processing elements p. In many cases, T_o grows sub linearly with respect to T_S. In such cases, we can see that efficiency increases if the problem size is increased keeping the number of processing elements constant. For such algorithms, it should be possible to keep the efficiency fixed by increasing both the size of the problem and the number of processing elements simultaneously. For instance, in Table 4.1, the efficiency of adding 64 numbers using four processing elements is 0.80. If the number of processing elements is increased to 8 and the size of the problem is scaled up to add 192 numbers, the efficiency remains 0.80. Increasing p to 16 and n to 512 results in the same efficiency. This ability to maintain efficiency at a fixed value by simultaneously increasing the number of processing elements and the size of the problem is exhibited by many parallel systems. We call such systems ***scalable*** parallel systems. The ***scalability*** of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements. It reflects a parallel system's ability to utilize increasing processing resources effectively.

Table 4.1. Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	*0.80*	0.57	0.33	0.17
192	1.0	0.92	*0.80*	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	*0.80*	0.62

Recall from Section 4.2.5 that a cost-optimal parallel system has an efficiency of $\Theta(1)$. Therefore, scalability and cost-optimality of parallel systems are related. A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately. For instance, Example 4.10 shows that the parallel system for adding n numbers on p processing elements is cost-optimal when $n = W(p \log p)$. Example 4.13 shows that the same parallel system is scalable if n is increased in proportion to $Q(p \log p)$ as p is increased.

Example 4.13 Scalability of adding n numbers

For the cost-optimal addition of n numbers on p processing elements $n = W(p \log p)$. As shown in Table 4.1, the efficiency is 0.80 for $n = 64$ and $p = 4$. At this point, the relation between n and p is $n = 8 p \log p$. If the number of processing elements is increased to eight, then $8 p \log p = 192$. Table 4.1 shows that the efficiency is indeed 0.80 with $n = 192$ for eight processing elements. Similarly, for $p = 16$, the efficiency is 0.80 for $n = 8 p \log p = 512$. Thus, this parallel system remains cost-optimal at an efficiency of 0.80 if n is increased as $8 p \log p$.

4.4.2 The Isoefficiency Metric of Scalability

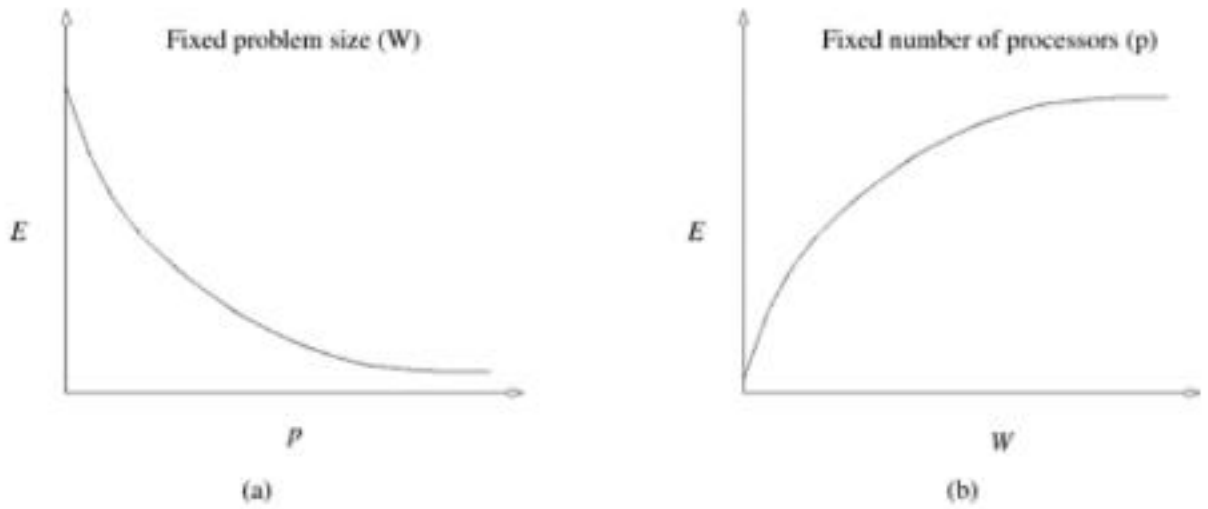
We summarize the discussion in the section above with the following two observations:

1. For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down. This phenomenon is common to all parallel systems.
2. In many cases, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

These two phenomena are illustrated in Figure 4.9(a) and (b), respectively. Following from these two observations, we define a scalable parallel system as one in which the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is also increased. It is useful to determine the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed. For different parallel systems, the problem size must increase at different rates in order to maintain a fixed efficiency as the number of processing elements is increased. This rate determines the degree of scalability of the

parallel system. As we shall show, a lower rate is more desirable than a higher growth rate in problem size. Let us now investigate metrics for quantitatively determining the degree of scalability of a parallel system. However, before we do that, we must define the notion of ***problem size*** precisely.

Figure 4.9. Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.



Problem Size When analyzing parallel systems, we frequently encounter the notion of the size of the problem being solved. Thus far, we have used the term ***problem size*** informally, without giving a precise definition. A naive way to express problem size is as a parameter of the input size; for instance, n in case of a matrix operation involving $n \times n$ matrices. A drawback of this definition is that the interpretation of problem size changes from one problem to another. For example, doubling the input size results in an eight-fold increase in the execution time for matrix multiplication and a four-fold increase for matrix addition (assuming that the conventional $\Theta(n^3)$ algorithm is the best matrix multiplication algorithm, and disregarding more complicated algorithms with better asymptotic complexities).

A consistent definition of the size or the magnitude of the problem should be such that, regardless of the problem, doubling the problem size always means performing twice the amount of computation. Therefore, we choose to express problem size in terms of the total number of basic operations required to solve the problem. By this definition, the problem size is $\Theta(n^3)$ for $n \times n$ matrix multiplication (assuming the conventional algorithm) and $\Theta(n^2)$ for $n \times n$ matrix addition. In order to keep it unique for a given problem, we define ***problem size*** as the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing element, where the best sequential algorithm is defined as in Section 4.2.3. Because it is defined in terms of sequential time complexity, the problem size is a function of the size of the input. The symbol we use to denote problem size is W.

In the remainder of this chapter, we assume that it takes unit time to perform one basic computation step of an algorithm. This assumption does not impact the analysis of any parallel system because the other hardware-related constants, such as message startup time, per-word

transfer time, and per-hop time, can be normalized with respect to the time taken by a basic computation step. With this assumption, the problem size W is equal to the serial runtime T_S of the fastest known algorithm to solve the problem on a sequential computer.

The Isoefficiency Function

Parallel execution time can be expressed as a function of problem size, overhead function, and the number of processing elements. We can write parallel runtime as:

Equation 4.10

$$T_P = \frac{W + T_o(W, p)}{p}$$

The resulng expression for speedup is

Equation 4.11

$$S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W, p)}.$$

Finally, we write the expression for eciency as

Equation 4.12

$$E = \frac{S}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + T_o(W, p)/W}.$$

In Equation 4.12, if the problem size is kept constant and p is increased, the efficiency decreases because the total overhead T_o increases with p. If W is increased keeping the number of processing elements fixed, then for scalable parallel systems, the efficiency increases. This is because T_o grows slower than $\Theta(W)$ for a fixed p. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.

For different parallel systems, W must be increased at different rates with respect to p in order to maintain a fixed efficiency. For instance, in some cases, W might need to grow as an exponential function of p to keep the efficiency from dropping as p increases. Such parallel systems are poorly scalable. The reason is that on these parallel systems it is difficult to obtain good speedups for a large number of processing elements unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly

scalable. That is because it can easily deliver speedups proportional to the number of processing elements for reasonable problem sizes.

For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio T_o/W in Equation 4.12 is maintained at a constant value. For a desired value E of efficiency,

Equation 4.13

$$\begin{aligned} E &= \frac{1}{1 + T_o(W, p)/W}, \\ \frac{T_o(W, p)}{W} &= \frac{1 - E}{E}, \\ W &= \frac{E}{1 - E} T_o(W, p). \end{aligned}$$

Let $K = E/(1 - E)$ be a constant depending on the efficiency to be maintained. Since T_o is a function of W and p, Equation 4.13 can be rewritten as

Equation 4.14

$$W = K T_o(W, p).$$

From Equation 4.14, the problem size W can usually be obtained as a function of p by algebraic manipulations. This function dictates the growth rate of W required to keep the efficiency fixed as p increases. We call this function the ***isoefficiency function*** of the parallel system. The isoefficiency function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements. A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processing elements, indicating that the parallel system is highly scalable. However, a large isoefficiency function indicates a poorly scalable parallel system. The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased.

Example 4.14 Isoefficiency function of adding numbers

The overhead function for the problem of adding n numbers on p processing elements is approximately $2p \log p$, as given by Equations 4 and 4. Substituting T_o by $2p \log p$ in Equation 4.14, we get

Equation 4.15

$$W = K 2p \log p.$$

Thus, the asymptotic isoefficiency function for this parallel system is $\Theta(p \log p)$. This means that, if the number of processing elements is increased from p to p', the problem size (in this case, n) must be increased by a factor of $(p' \log p')/(p \log p)$ to get the same efficiency as on p processing elements. In other words, increasing the number of processing elements by a factor of p'/p requires that n be increased by a factor of $(p' \log p')/(p \log p)$ to increase the speedup by a factor of p'/p.

In the simple example of adding n numbers, the overhead due to communication (hereafter referred to as the ***communication overhead***) is a function of p only. In general, communication overhead can depend on both the problem size and the number of processing elements. A typical overhead function can have several distinct terms of different orders of magnitude with respect to p and W. In such a case, it can be cumbersome (or even impossible) to obtain the isoefficiency function as a closed function of p. For example, consider a hypothetical parallel system for which $T_o = p^{3/2} + p^{3/4} W^{3/4}$. For this overhead function, Equation 4.14 can be rewritten as $W = Kp^{3/2} + Kp^{3/4} W^{3/4}$. It is hard to solve this equation for W in terms of p.

Recall that the condition for constant efficiency is that the ratio T_o/W remains fixed. As p and W increase, the efficiency is nondecreasing as long as none of the terms of T_o grow faster than W. If T_o has multiple terms, we balance W against each term of T_o and compute the respective isoefficiency functions for individual terms. The component of T_o that requires the problem size to grow at the highest rate with respect to p determines the overall asymptotic isoefficiency function of the parallel system. Example 4.15 further illustrates this technique of isoefficiency analysis.

Example 4.15 Isoefficiency function of a parallel system with a complex overhead function

Consider a parallel system for which $T_o = p^{3/2} + p^{3/4} W^{3/4}$. Using only the first term of T_o in Equation 4.14, we get

Equation 4.16

$$W = Kp^{3/2}.$$

Using only the second term, Equation 4.14 yields the following relation between W and p:

Equation 4.17

$$\begin{aligned} W &= Kp^{3/4} W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4 p^3 \end{aligned}$$

To ensure that the efficiency does not decrease as the number of processing elements increases, the first and second terms of the overhead function require the problem size to grow as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively. The asymptotically higher of the two rates, $\Theta(p^3)$, gives the overall asymptotic isoefficiency function of this parallel system, since it subsumes the rate dictated by

the other term. The reader may indeed verify that if the problem size is increased at this rate, the efficiency is $\Theta(1)$ and that any rate lower than this causes the efficiency to fall with increasing p.

In a single expression, the isoefficiency function captures the characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. After performing isoefficiency analysis, we can test the performance of a parallel program on a few processing elements and then predict its performance on a larger number of processing elements. However, the utility of isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processing elements. Section 4.4.5shows how the isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm. We will see in Chapter 13 that isoefficiency analysis can also be used to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processing elements and communication channels. Chapter 11 illustrates how isoefficiency analysis can be used even for parallel algorithms for which we cannot derive a value of parallel runtime.

4.4.3 Cost-Optimality and the Isoefficiency Function

In Section 4.2.5, we stated that a parallel system is cost-optimal if the product of the number of processing elements and the parallel execution time is proportional to the execution time of the fastest known sequential algorithm on a single processing element. In other words, a parallel system is cost-optimal if and only if

Equation 4.18

$$pT_P = \Theta(W).$$

Substituting the expression for T_P from the right-hand side of Equation 4.10, we get the following:

Equation 4.19

$$\begin{aligned} W + T_o(W, p) &= \Theta(W) \\ T_o(W, p) &= O(W) \end{aligned}$$

Equation 4.20

$$W = \Omega(T_o(W, p))$$

Equations 4 and 4 suggest that a parallel system is cost-optimal if and only if its overhead function does not asymptotically exceed the problem size. This is very similar to the condition given by Equation 4.14 for maintaining a fixed efficiency while increasing the number of processing elements in a parallel system. If Equation 4.14 yields an isoefficiency function $f(p)$, then it follows from Equation 4.20 that the relation $W = W(f(p))$ must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up. The following example further illustrates the relationship between cost-optimality and the isoefficiency function.

Example 4.16 Relationship between cost-optimality and isoefficiency

Consider the cost-optimal solution to the problem of adding n numbers on p processing elements, presented in Example 4.10. For this parallel system, $W \approx n$, and $T_o = \Theta(p \log p)$. From Equation 4.14, its isoefficiency function is $\Theta(p \log p)$; that is, the problem size must increase as $\Theta(p \log p)$ to maintain a constant efficiency. In Example 4.10 we also derived the condition for cost-optimality as $W = W(p \log p)$.

4.4.4 A Lower Bound on the Isoefficiency Function

We discussed earlier that a smaller isoefficiency function indicates higher scalability. Accordingly, an ideally-scalable parallel system must have the lowest possible isoefficiency function. For a problem consisting of W units of work, no more than W processing elements can be used cost-optimally; additional processing elements will be idle. If the problem size grows at a rate slower than $\Theta(p)$ as the number of processing elements increases, then the number of processing elements will eventually exceed W. Even for an ideal parallel system with no communication, or other overhead, the efficiency will drop because processing elements added beyond $p = W$ will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\Theta(p)$ to maintain fixed efficiency; hence, $W(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable parallel system is $\Theta(p)$.

4.4.5 The Degree of Concurrency and the Isoefficiency Function

A lower bound of $W(p)$ is imposed on the isoefficiency function of a parallel system by the number of operations that can be performed concurrently. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its ***degree of concurrency***. The degree of concurrency is a measure of the number of operations that an algorithm can perform in parallel for a problem of size W; it is independent of the parallel architecture. If $C(W)$ is the degree of concurrency of a parallel algorithm, then for a problem of size W, no more than $C(W)$ processing elements can be employed effectively.

Example 4.17 Effect of concurrency on isoefficiency function

Consider solving a system of n equations in n variables by using Gaussian elimination. The total amount of computation is $\Theta(n^3)$. But then variables must be eliminated one after the other, and eliminating each variable requires $\Theta(n^2)$ computations. Thus, at most $\Theta(n^2)$ processing elements can be kept busy at any time. Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\Theta(W^{2/3})$ and at most $\Theta(W^{2/3})$ processing elements can be used efficiently. On the other hand, given p processing elements, the problem size should be at least $W(p^{3/2})$ to use them all. Thus, the isoefficiency function of this computation due to concurrency is $\Theta(p^{3/2})$. The isoefficiency function due to concurrency is optimal (that is, $\Theta(p)$) only if the degree of concurrency of the parallel algorithm is $\Theta(W)$. If the degree of concurrency of an algorithm is less than $\Theta(W)$, then the isoefficiency function due to concurrency is worse (that is, greater) than $\Theta(p)$. In such cases, the overall isoefficiency function of a parallel system is given by the

maximum of the isoefficiency functions due to concurrency, communication, and other overheads.

4.5 Minimum Execution Time and Minimum Cost-Optimal Execution Time

We are often interested in knowing how fast a problem can be solved, or what the minimum possible execution time of a parallel algorithm is, provided that the number of processing elements is not a constraint. As we increase the number of processing elements for a given problem size, either the parallel runtime continues to decrease and asymptotically approaches a minimum value, or it starts rising after attaining a minimum value (Problem 4.12). We can determine the minimum parallel runtime T_P^{min} for a given W by differentiating the expression for T_P with respect to p and equating the derivative to zero (assuming that the function $T_P(W, p)$ is differentiable with respect to p). The number of processing elements for which T_P is minimum is determined by the following equation:

Equation 4.21

$$\frac{d}{dp}T_P = 0$$

Let p_0 be the value of the number of processing elements that satisfies Equation 4.21. The value of T_P^{min} can be determined by substituting p_0 for p in the expression for T_P. In the following example, we derive the expression for T_P^{min} for the problem of adding n numbers.

Example 4.18 Minimum execution time for adding n numbers

Under the assumptions of Example 4.12, the parallel run time for the problem of adding n numbers on p processing elements can be approximated by

Equation 4.22

$$T_P = \frac{n}{p} + 2\log p.$$

Equating the derivative with respect to p of the right-hand side of Equation 4.22 to zero we get the solutions for p as follows:

Equation 4.23

$$\begin{aligned} -\frac{n}{p^2}+\frac{2}{p} &= 0 \\ -n+2p &= 0 \\ p &= \frac{n}{2} \end{aligned}$$

Substituting $p = n/2$ in Equation 4.22, we get

Equation 4.24

$$T_P^{min} = 2\log n.$$

In Example 4.18, the processor-time product for $p = p_0$ is $\Theta(n \log n)$, which is higher than the $\Theta(n)$ serial complexity of the problem. Hence, the parallel system is not cost-optimal for the value of p that yields minimum parallel runtime. We now derive an important result that gives a lower bound on parallel runtime if the problem is solved cost-optimally.

Let $T_P^{cost\_opt}$ be the minimum time in which a problem can be solved by a cost-optimal parallel system. From the discussion regarding the equivalence of cost-optimality and the isoefficiency function in Section 4.4.3, we conclude that if the isoefficiency function of a parallel system is $\Theta(f(p))$, then a problem of size W can be solved cost-optimally if and only if $W = W(f(p))$. In other words, given a problem of size W, a cost-optimal solution requires that $p = O(f^{-1}(W))$. Since the parallel runtime is $\Theta(W/p)$ for a cost-optimal parallel system (Equation 4.18), the lower bound on the parallel runtime for solving a problem of size W cost-optimally is

Equation 4.25

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right).$$

Example 4.19 Minimum cost-optimal execution time for adding n numbers

As derived in Example 4.14, the isoefficiency function $f(p)$ of this parallel system is $\Theta(p \log p)$. If $W = n = f(p) = p \log p$, then $\log n = \log p + \log \log p$. Ignoring the double logarithmic term, $\log n \approx \log p$. If $n = f(p) = p \log p$, then $p = f^{-1}(n) = n/\log p \approx n/\log n$. Hence, $f^{-1}(W) = \Theta(n/\log n)$. As a consequence of the relation between cost-optimality and the isoefficiency function, the maximum number of processing elements that can be used to solve this problem cost-optimally is $\Theta(n/\log n)$. Using $p = n/\log n$ in Equation 4.2, we get

Equation 4.26

$$T_P^{cost\_opt} = \log n + \log\left(\frac{n}{\log n}\right) = 2\log n - \log\log n.$$

It is interesting to observe that both T_P^{min} and $T_P^{cost\_opt}$ for adding n numbers are $\Theta(\log n)$ (Equations 4.24 and 4.26). Thus, for this problem, a cost-optimal solution is also the asymptotically fastest solution. The parallel execution time cannot be reduced asymptotically by using a value of p greater than that suggested by the isoefficiency function for a given problem size (due to the equivalence between cost-optimality and the isoefficiency function). This is not true for parallel systems in general, however, and it is quite possible that $T_P^{cost\_opt} > \Theta(T_P^{min})$. The following example illustrates such a parallel system.

Example 4.20 A parallel system with

Consider the hypothetical parallel system of Example 4.15, for which

Equation 4.27

$$T_o = p^{3/2} + p^{3/4}W^{3/4}.$$

From Equation 4.10, the parallel runtime for this system is

Equation 4.28

$$T_P = \frac{W}{p} + p^{1/2} + \frac{W^{3/4}}{p^{1/4}}.$$

Using the methodology of Example 4.18,

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}p}T_P = -\frac{W}{p^2} + \frac{1}{2p^{1/2}} - \frac{W^{3/4}}{4p^{5/4}} &= 0, \\
-W + \frac{1}{2}p^{3/2} - \frac{1}{4}W^{3/4}p^{3/4} &= 0, \\
p^{3/4} &= \frac{1}{4}W^{3/4} \pm \left(\frac{1}{16}W^{3/2} + 2W\right)^{1/2} \\
&= \Theta(W^{3/4}), \\
p &= \Theta(W).
\end{aligned}$$

From the preceding analysis, $p_0 = \Theta(W)$. Substituting p by the value of p_0 in Equation 4.28, we get

Equation 4.29

$$T_P^{min} = \Theta(W^{1/2}).$$

According to Example 4.15, the overall isoefficiency function for this parallel system is $\Theta(p^3)$, which implies that the maximum number of processing elements that can be used cost-optimally is $\Theta(W^{1/3})$. Substituting $p = \Theta(W^{1/3})$ in Equation 4.28, we get

Equation 4.30

$$T_P^{cost\_opt} = \Theta(W^{2/3}).$$

A comparison of Equations 4.29 and 4.30 shows that $T_P^{cost\_opt}$ is asymptotically greater than T_P^{min}.

In this section, we have seen examples of both types of parallel systems: those for which $T_P^{cost\_opt}$ is asymptotically equal to T_P^{min}, and those for which $T_P^{cost\_opt}$ is asymptotically greater than T_P^{min}. Most parallel systems presented in this book are of the first type. Parallel systems for which the runtime can be reduced by an order of magnitude by using an asymptotically higher number of processing elements than indicated by the isoefficiency function are rare.

While deriving the minimum execution time for any parallel system, it is important to be aware that the maximum number of processing elements that can be utilized is bounded by the degree of concurrency $C(W)$ of the parallel algorithm. It is quite possible that p_0 is greater than $C(W)$ for a parallel system (Problems 4.13 and 4.14). In such cases, the value of p_0 is meaningless, and T_P^{min} is given by

Equation 4.31

$$T_P^{min} = \frac{W + T_o(W, C(W))}{C(W)}.$$

4.6 Dense Matrix Algorithms

- A matrix is said to be dense, if it has maximum number of non zero entries. If the number of zeroes is greater than non zero entries then it is called sparse matrix. Both these kinds of matrices must have different types of algorithms to be processed efficiently.
- In this section, we discuss parallel algorithms for dense matrices. The characteristic of these algorithms is, each process has one task, so that decomposition of the process is not required. This simplifies the algorithms and its processing.

4.6.1 Matrix Vector Multiplication

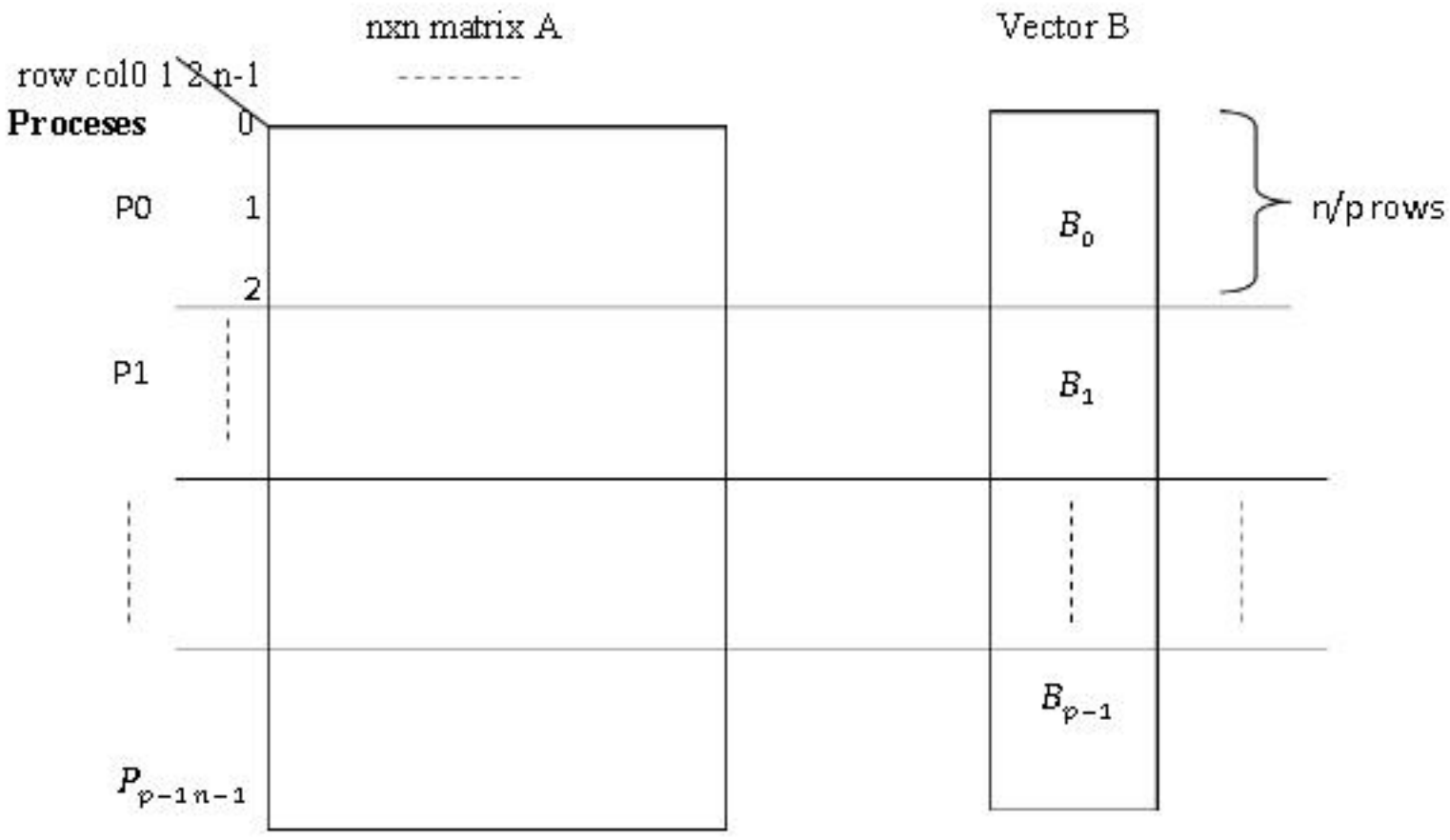
- Vector is a matrix with one column. Here we consider the problem of multiplying a n x n matrix with n x 1 matrix which is a vector. Both these matrices are dense. The product is again a vector.
- Let A be a dense matrix with n x n dimension and B be a vector with n rows. The result is C with n rows and 1 columns. To start, let us consider the sequential algorithms. It is simple matrix multiplication where second matrix is known to have only one column.

```
int *mat_vect_mult_Seq (int A[n][n], int B[n])
{
        int c[n];
        for(i=0;i<n;i++)
        {
                c[i] = 0;
                for(j=0;j<n;j++)
                c[i] = c[i] + A[i][j] * B[j];
        }return c;
}
```

- The complexity of this sequential algorithms is $O(n^2)$. The following subsections discuss the parallel algorithms.

4.6.1.1 Row-Wise 1-D Partitioning

- The first step to multiply matrix and vector is to partition the rows in the matrix into p parts where, p is the number of available processors. Let us assume that $p|n$ (i.e. completely divides n or n mod p = 0).
- Thus each process P_i where $0 \le i \le p-1$ gets n/p rows from matrix A and n/p elements from vector B. Fig. 4.6.1 shows this.



Bi is the small vector with n/p elements assigned to Pi

Fig. 4.6.1 Initial partition of matrix and vector

- In the next step, by using one of all broadcast, the vector elements allocated to one process, is broadcasted to all other processes. Thus each process will now have n/p rows of matrix A and all elements of vector B which are partitioned into p blocks.
- Fig 4.6.2 shows vector broadcasting and Fig 4.6.3 gives the final allocation to the process, and product C. Now we consider two cases where p=n and n>p.

Process vector blocks :

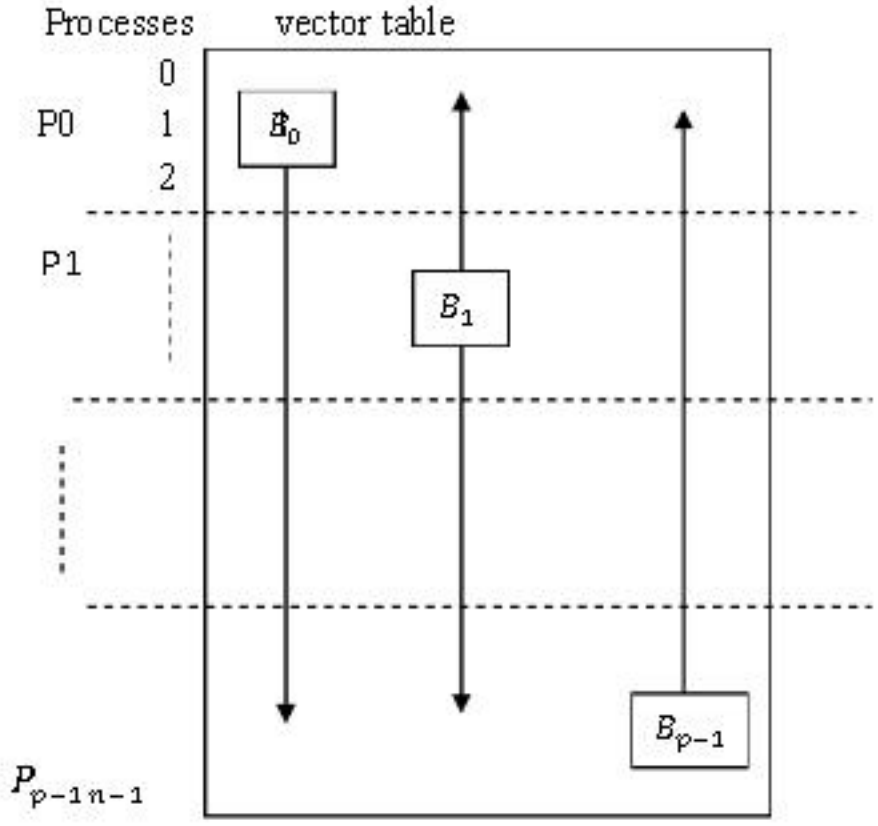


Fig. 4.6.2 Vector Broadcasting

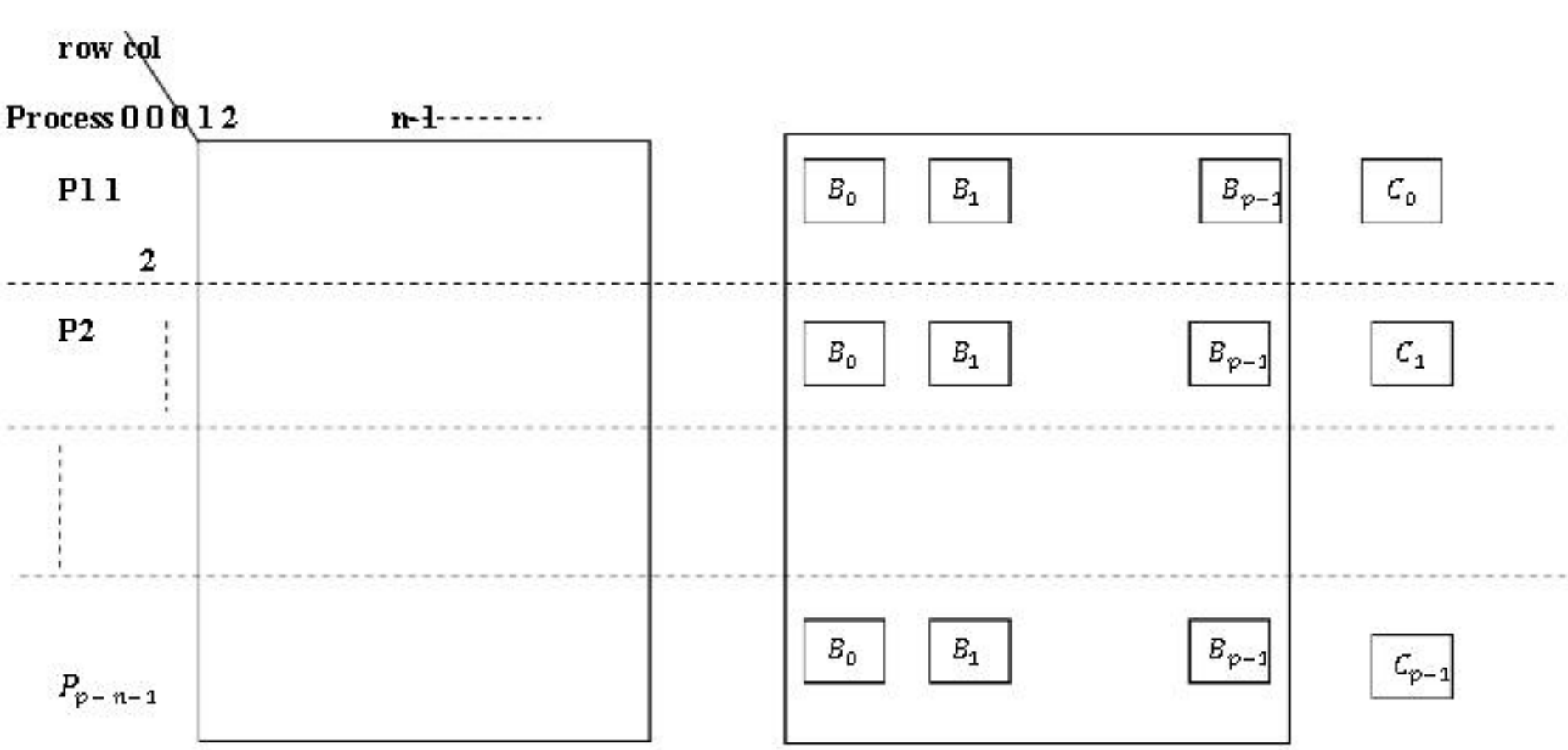


Fig. 4.6.3

Case 1 : Number of processes = number of rows i.e. p = n

In this case each process P_i, gets ith row of matrix A. The process computing ith element in the resultant vector C by using the following equation.

$$C[i] = \sum_{j=0}^{n-1} A[i][j] * B[j]$$

Complexity of the algorithm

Broadcasting of single element of vector B to all processes, requires O(n) time. Within a process, one row of A is multiplied to vector B, which requires O(n) time. Thus the time complexity is O(n).

Case 2 : Number of processes < number of rows i.e. p<n

In this case, each process gets n/p rows of matrix A. Every process must multiply these n/p rows with vector B, to produce n/p elements of the product vector C.

Time complexity :

In this case, the type of broadcasting is all to all. All n/p elements of vector B_i in P_i must be broadcasted to remaining processes. This leads to the time complexity of $t_s log_p + t_w$ (n/p)(p-1) where, t_s = message startup time; t_w = per-word transfer time.

if p is very large, then it can be written as $t_s \log p + t_{wn}$.

After broadcasting, each process multiplies n/p row with the vector. This takes n^2/p time complexity.

4.6.1.2 2-D Partitioning

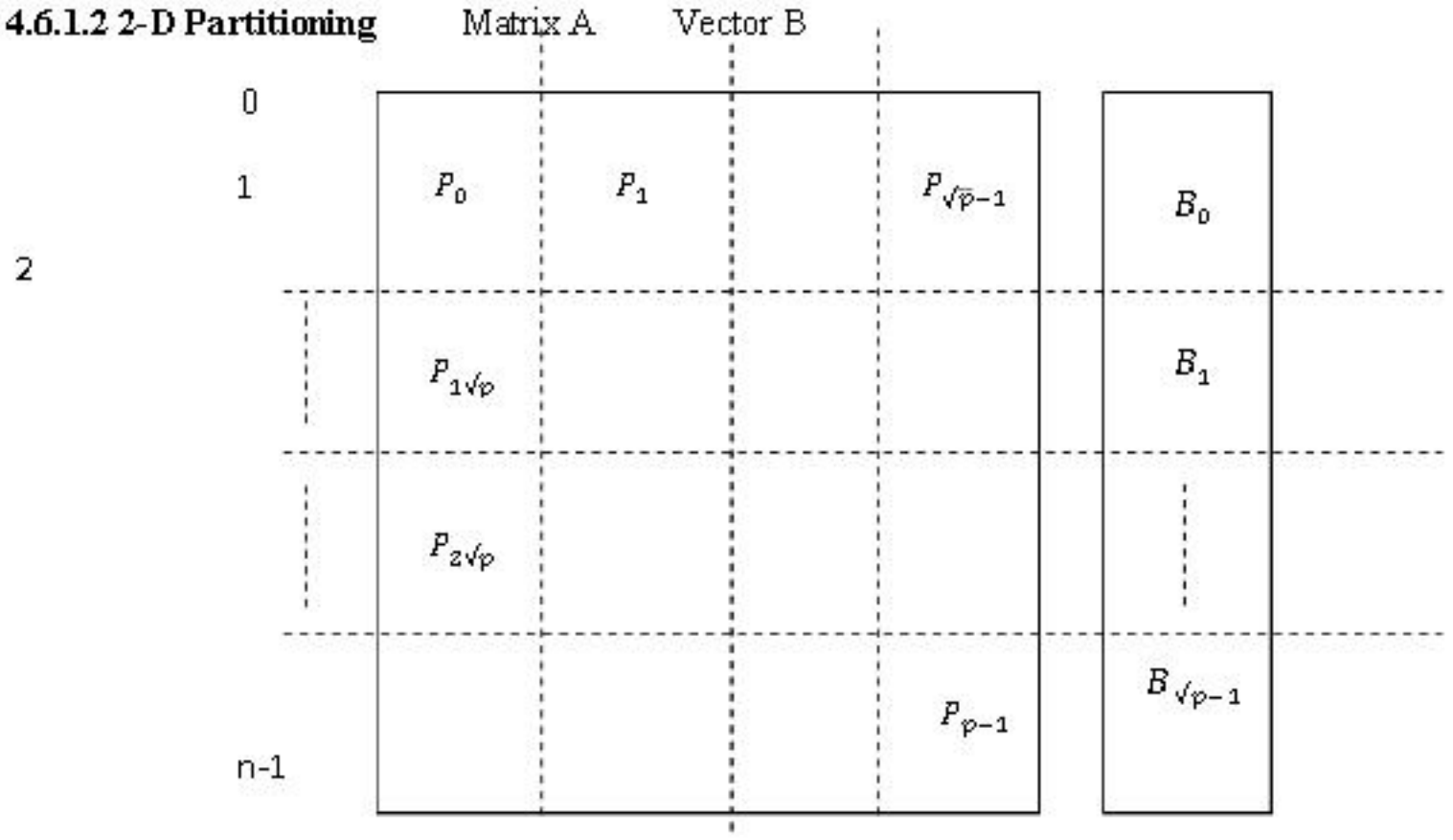


Fig. 4.6.4 Initial partition of matrix and vector

- In this method, the matrix is distributed among the processes using a block 2-D partitioning. In the previous section, a process had a complete row of the matrix. Now the matrix is divided into blocks of 2-dimension and allocated to one process. This is shown in fig.4.6.4.
- The number of processes p must be a perfect square, so that, nxn elements of matrix A can be equally distributed to P processes. The vector B is partitioned into $\sqrt{P}$ number of blocks.
- A block B, where $0 \leq i \leq \sqrt{P}-1$, allocated to the process $P_{i*\sqrt{p}+i}$. This is shown in fig 4.1.5

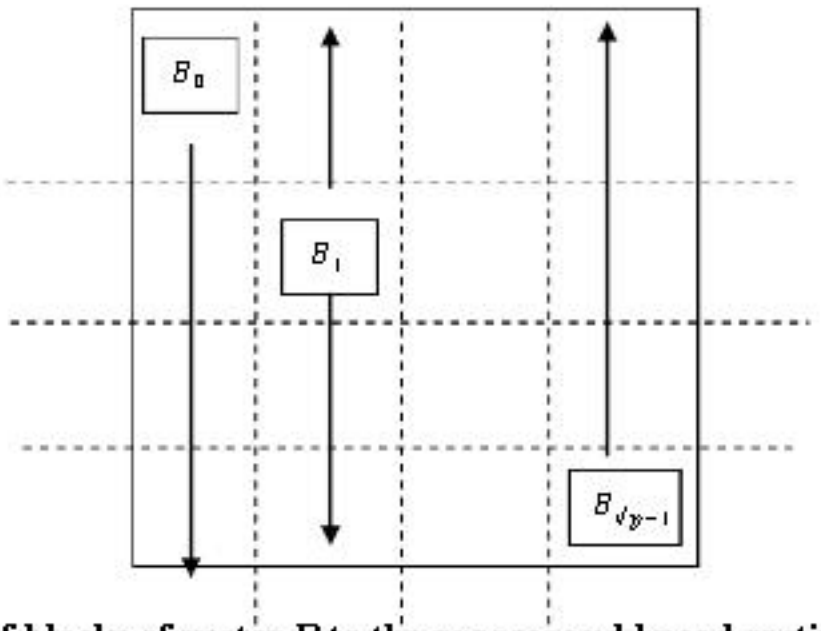


Fig. 4.6.5 Allocation of blocks of vector B to the process and broad casting them through column

- After allocating B_i to $P_{i*\sqrt{p}+i}$, this block must be broadcasted to all the processes in the column i. Fig 4.6.5 also shows this.

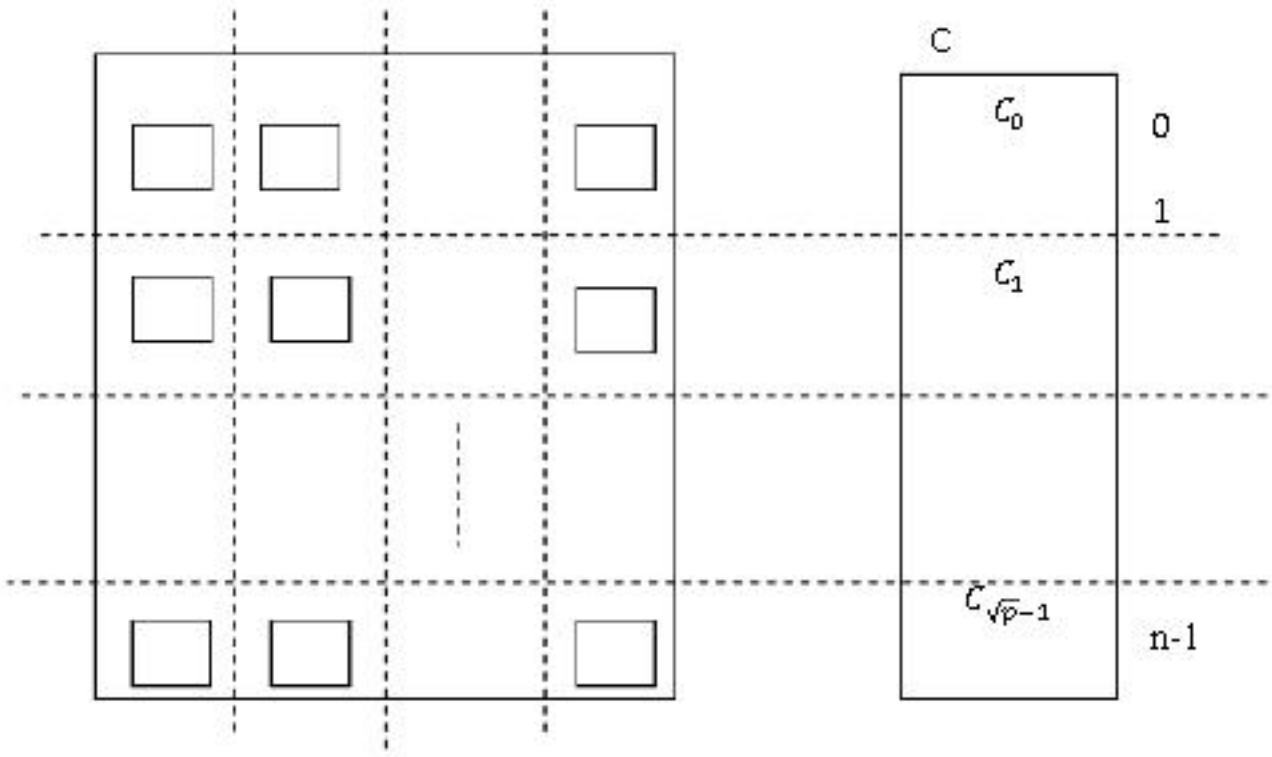
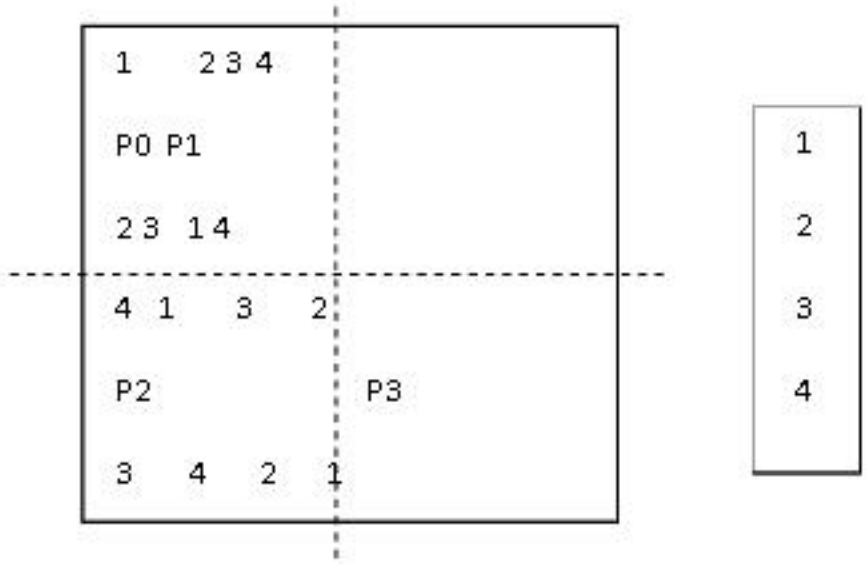


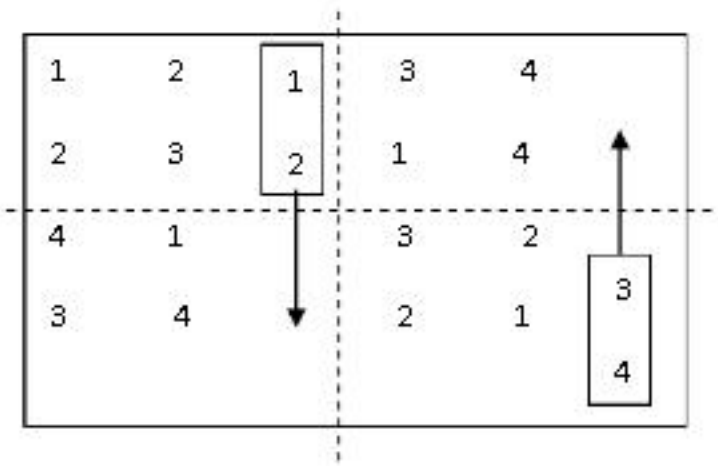
Fig. 4.6.6

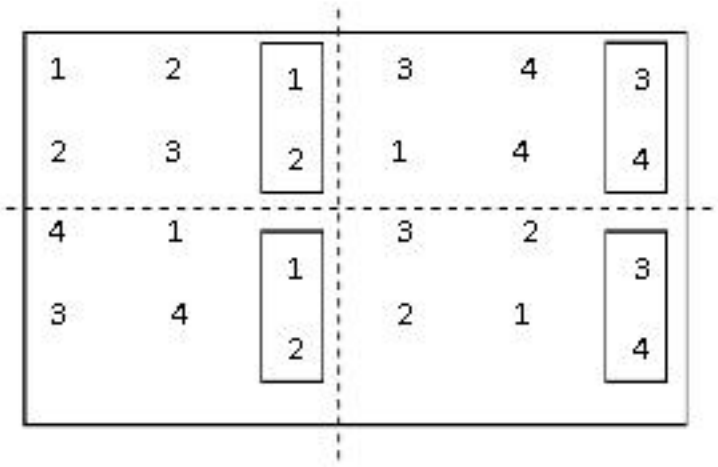
- Now each process has a $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$ matrix and $\sqrt{p}$ vector. These are multiplies using sequential algorithm. In the next step, these partial results are combined by using all-in-one reduction. All columns are added to get the resultant vector. See Fig 4.6.6.
- Fig. 4.6.7 shows the working of this method, with the help of an example. In this example. 4 x 4matrix is multiplied to 4x1 vector. The number of processes is 4.
- Further, we discuss two cases that were discussed in 1-D partitioning.

Case 1 : One element per process, i.e. P = n^2.

- Using n^2 number of processes, each process will be allotted with one element from matrix A, and one element from vector B. Thus each process does a single multiplication which has O(1) time complexity. Before this, one to all broadcasting is required, which has the complexity of O(n).







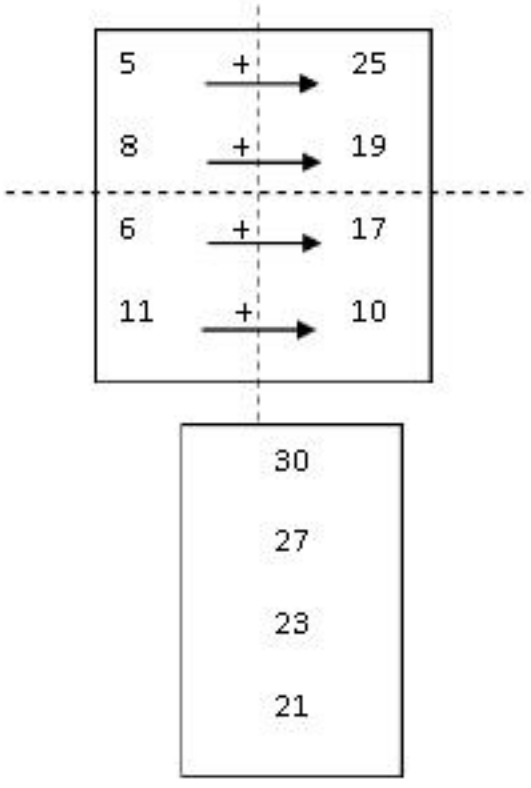


Fig. 4.6.7 Example of matrix-vector multiplication with 2-D partitioning

The all-to-one reduction also has O(n) time complexity. Thus by using n^2 processes, the multiplication is done in O(n) time complexity.

Case 2 : Using $p < n^2$ processes

Here each process is allotted with $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$ block of matrix A. The vector is partitioned into $\sqrt{p}$ parts, each part with $\frac{n}{\sqrt{p}}$ elements. This allocation has time complexity of $t_s + t_w \frac{n}{\sqrt{p}}$. The one-to-all broadcasting is done with time complexity $(t_s + t_w \frac{n}{\sqrt{p}}) \log(\sqrt{p})$. The all-to-one reduction also has the same complexity.

4.6.2 Matrix-Matrix Multiplication

WE consider two nxn matrices A and B that multiplied to give the product c, again an nxn matrix. The sequential algorithm takes time complexity o fO(n^3). Following is the algorithm

```
int **mat-mat-mul-seq (int A[n][n],int B[n][n])
{       int c[n][n]
        for(i=0;i<n;i++)
        {       for (j=0;j<n;j++)
                {       c[i][j] = 0
                        for (k=0;k<n;k++)
                        c[i][j] = c[i][j]+A[i][k] + B[k][j];
                }
        } return c; }
```

4.6.2.1 A Simple Parallel Algorithm

We discuss a simple parallel algorithm for matrix multiplication. The nxn matrices A and B are partitioned into P blocks of size $\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}$ each.

- These blocks are allotted to p processes. This is similar to 2-D partitioning, discussed earlier. For simplicity, we consider the processor in a two-dimensional notation and are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$.
- A process $P_{i,j}$ is allotted with the $A_{i,j}$ and $B_{i,j}$ blocks, where 0≤ i, j≤ √p-1. Fig 4.6.8 shows this now in each row of processes, an all-to-one broadcast of matrix B is done. This results into the process $P_{i,j}$ to have $A_{i,0}$, $A_{i,1}$, ... $A_{i,\sqrt{p}-1}$ and $B_{0,j}$, $B_{1,j}$, ...$B_{\sqrt{p}-1,j}$ blocks.
- Blocks of A are shifted through the rows and B are shifted through columns. See fig. 4.6.11

Matrix A

	0	1	2		n-1
{	$A_{0,0}$	$A_{0,1}$		-----	$A_{0,\sqrt{p}-1}$
	$A_{1,0}$	$A_{1,1}$		-----	$A_{1,\sqrt{p}-1}$
	⋮	⋮		⋮	⋮
	$A_{\sqrt{p}-1,0}$	$A_{\sqrt{p}-1,1}$		------	$A_{\sqrt{p}-1,\sqrt{p}-1}$

Matrix B

0	1	2		n-1
$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	------	$B_{0,\sqrt{p}-1}$
$B_{1,0}$	$B_{1,0}$	$B_{1,2}$	-----	$B_{1,\sqrt{p}-1}$
⋮	⋮			⋮
$B_{\sqrt{p}-1,0}$	$B_{\sqrt{p}-1,1}$	$B_{\sqrt{p}-1,2}$	------	$B_{\sqrt{p}-1,\sqrt{p}-1}$

$P_{0,0}$ $A_{0,0}$ $B_{0,0}$	$P_{0,1}$ $A_{0,1}$ $B_{0,1}$	---------	$P_{0\sqrt{p}-1}$ $A_{0\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}$
$P_{1,0}$ $A_{1,0}$ $B_{1,0}$	$P_{1,1}$ $A_{1,1}$ $B_{1,1}$	---------	$P_{1\sqrt{p}-1}$ $A_{1\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}$
⋮	⋮		⋮
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,0}$ $B_{\sqrt{p}-1,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,1}$ $B_{\sqrt{p}-1,1}$	---------	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.6.8 Initial allotment of blocks of matrices A and B to processes

$P_{0,0}$	$P_{0,1}$	$P_{0,\sqrt{p}-1}$
$A_{0,0}, A_{0,1}, \ldots, A_{0,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots, B_{\sqrt{p}-1,0}$	$A_{0,0}, A_{0,1}, \ldots, A_{0,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots, B_{\sqrt{p}-1,1}$	$A_{0,0}, A_{0,1}, \ldots, A_{0,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots, B_{\sqrt{p}-1,\sqrt{p}-1}$
$P_{0,0}$ $A_{0,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots, B_{\sqrt{p}-1,0}$	$P_{1,1}$ $A_{1,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots, B_{\sqrt{p}-1,1}$	$P_{1,\sqrt{p}-1}$ $A_{1,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots, B_{\sqrt{p}-1,\sqrt{p}-1}$
⋮	⋮	⋮
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,0}, A_{\sqrt{p}-1,1}, \ldots, A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{0,0}, B_{1,0}, \ldots, B_{\sqrt{p}-1,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,0}, A_{\sqrt{p}-1,1}, \ldots, A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{0,1}, B_{1,1}, \ldots, B_{\sqrt{p}-1,1}$	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{1,0}, A_{1,1}, \ldots, A_{1,\sqrt{p}-1}$ $B_{0,\sqrt{p}-1}, B_{1,\sqrt{p}-1}, \ldots, B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.6.9 Broadcasting A's block through rows and B's blocks through columns

- Now, each process $P_{i,j}$ computes the resulting sub block $C_{i,j}$ as follows:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$$

The multiplication of sub blocks can be done by using sequential algorithm discussed earlier.

The all-to-all broadcasting through each row and column requires, $2(t_s \log\sqrt{p} + t_w (n^2/(\sqrt{p})(\sqrt{p}-1))$ time complexity. The multiplication of sub matrices of size $\left(\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}}\right)$ require $(n^3/(\sqrt{p})^3 = n^3/(p\sqrt{p})$. Each process has to do such $\sqrt{p}$ multiplication.

$$\text{Complexity} = \sqrt{p} * n^3/(p\sqrt{p}) = n^3/p$$

Thus total time complexity is given by $\frac{n^3}{p} + t_s \log p + 2t_w \frac{n^3}{\sqrt{p}}$

4.6.2.2 Cannon's Algorithm

- Cannon's algorithm works similar to the simple parallel algorithm discussed in the previous section. But is more memory efficient. The initial assignment of the block of matrices A and B to the process is as shown in Fig. 4.6.10.
- In the previous algorithm , the blocks of A were broadcasted though the rows and blocks of B were broadcasted through the columns. In cannon's algorithm, instead of broadcasting, the blocks are shifted.
- Blocks of A are shifted through the rows and B are shifted through columns. See Fig. 4.6.11

- While assigning the blocks to the processes, the row I of blocks in A are shifted i blocks and columns i of blocks in A re shifted i blocks.
- The initial allotment, the process is shown in Fig 4.6.11.
- Thus after initial allocation, the process $P_{i,j}$ has modifies blocks $A_{x,y}$ and $B_{y,z}$. It will find the partial result by multiplying $A_{x,y}$ and $B_{y,z}$ and stored in $C_{i,j}$.
- After first shift, the process $P_{i,j}$ has $A_{x,(y+1)}$ mod √P and $B_{(x+1)}$ mod √P, y.
- Now multiply these blocks and add the result to $C_{i,j}$. This process is continued for √P times. This is shown in fir. 4.6.12
- While assigning the blocks to the processes, the row i of blocks in A one shifted i blocks, and column i of blocks in A one shifted i blocks.
- The initial allotment to the processes is shown in fig. 4.6.11.
- Blocks of A are shifted through the rows and B are shifted through columns. See fig. 4.6.11

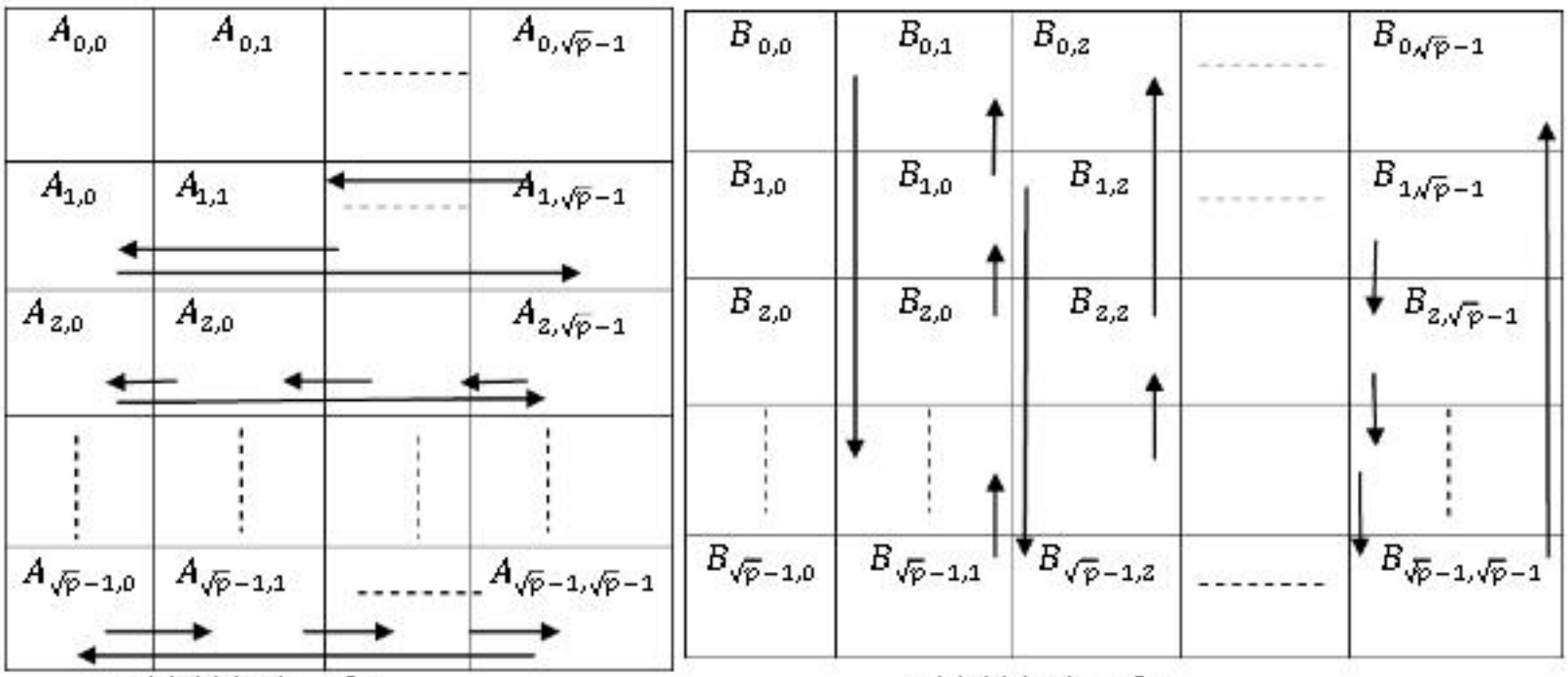


Fig. 4.6.10 The initial assignment of the block of matrices A and B to the process

$P_{0,0}$ $A_{0,0}$ $B_{0,0}$	$P_{0,1}$ $A_{0,1}$ $B_{1,1}$	$P_{0,2}$ $A_{0,2}$ $B_{0,2}$	---------	$P_{0,\sqrt{p}-1}$ $A_{0,\sqrt{p}-1}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
$P_{1,0}$ $A_{1,1}$ $B_{1,0}$	$P_{1,1}$ $A_{1,2}$ $B_{2,1}$	$P_{1,2}$ $A_{1,3}$ $B_{3,2}$	---------	$P_{1,\sqrt{p}-1}$ $A_{1,0}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
$P_{2,0}$ $A_{2,2}$ $B_{2,0}$	$P_{2,1}$ $A_{2,3}$ $B_{3,1}$	$P_{2,2}$ $A_{2,2}$ $B_{4,2}$	---------	$P_{2,\sqrt{p}-1}$ $A_{2,1}$ $B_{1,\sqrt{p}-1}$
⋮	⋮	⋮		⋮
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{\sqrt{p}-1,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,0}$ $B_{0,1}$	$P_{\sqrt{p}-1,2}$ $A_{\sqrt{p}-1,1}$ $B_{1,2}$	---------	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{\sqrt{p}-1,\sqrt{p}-2}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.6.11 Initial allotment to the processes

$P_{0,0}$ $A_{0,1}$ $B_{1,0}$	$P_{0,1}$ $A_{0,2}$ $B_{2,1}$	$P_{0,2}$ $A_{0,3}$ $B_{3,2}$	---------	$P_{0,\sqrt{p}-1}$ $A_{0,0}$ $B_{0,\sqrt{p}-1}$
$P_{1,0}$ $A_{1,2}$ $B_{2,0}$	$P_{1,1}$ $A_{1,3}$ $B_{3,1}$	$P_{1,2}$ $A_{1,4}$ $B_{4,2}$	---------	$P_{1,\sqrt{p}-1}$ $A_{1,1}$ $B_{1,\sqrt{p}-1}$
$P_{2,0}$ $A_{2,3}$ $B_{3,0}$	$P_{2,1}$ $A_{2,4}$ $B_{4,1}$	$P_{2,2}$ $A_{2,5}$ $B_{5,2}$	---------	$P_{2,\sqrt{p}-1}$ $A_{2,2}$ $B_{2,\sqrt{p}-1}$
⋮	⋮	⋮		⋮
$P_{\sqrt{p}-1,0}$ $A_{\sqrt{p}-1,0}$ $B_{0,0}$	$P_{\sqrt{p}-1,1}$ $A_{\sqrt{p}-1,1}$ $B_{1,1}$	$P_{\sqrt{p}-1,2}$ $A_{\sqrt{p}-1,2}$ $B_{2,2}$	---------	$P_{\sqrt{p}-1,\sqrt{p}-1}$ $A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$

Fig. 4.6.12 Blocks of A and B after first shift

- The arrow lines show the next shift. This is repeated $\sqrt{p}$ times.
- The time complexity of Cannon's algorithm is same as that of simple parallel algorithm. But when space is considered, in the simple algorithm each process had to score $\sqrt{p}$ blocks of A and $\sqrt{p}$ blocks of B. Where as in connon's algorithm, at any given time, a process stores one block of A and one block of B.

4.6.2.3 The DNS Algorithm

- In the simple parallel algorithm and in Cannon's algorithm, we can use maximum n^2 processes. In this section, we discuss an algorithm, in which n^3 processes can be used. This algorithm is designed by Eliezer Dekel, David Nassimi, and Sartaj Sahni. Hence the name of the algorithm is DNS algorithm.
- Here we have n^3 processes logically arranged in a three dimensional nxnxx array. The basic idea behind this algorithm is, the sequential algorithm does n^3 multiplication, that can be assigned to n^3 processors.
- The processes are labeled as per their logical location as, $P_{i,j,k}$, where 0≤ I, j,k ≤ n-1. The elements $A_{i,k}$ and $B_{k,j}$ are allocated to $P_{i,j,k}$. These elements are multiplied. Now the results produced by $P_{i,j,0}$, $P_{i,j,1}$, ……., $P_{i,j,n-1}$ are added to find $C_{i,j}$.
- The multiplication is done in O(1) time and the additions can be done parallel in O(log n) time complexity.
- Thus the time complexity of DNS algorithm using n^3 processes is O(log n).
- Let us consider two 4x4 matrixes for simplicity. The DNS algorithm requires $4^3 = 64$ processes.
- Fig. 4.1.13 shows distribution of processes and Fig 4.1.14 shows allocation of the matrices. Initially the matrix one stored in processes with k=0. The kth row of B and kth column of A are moved to kth plane.
- The rows of B and columns of A are broadcasted to the complete k plane. It is shown in Fig 4.1.15.
- Let result of multiplication of two elements allotted to each processors $P_{i,j,k}$ be $r_{i,j,k}$ the product matrix C is calculated as follows :

$$c_{i,j} = \sum_{k=0}^{n-1} r_{i,j,k}$$

- Let us consider two matrices A and B given by

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 0 & 4 & 3 \\ 1 & 4 & 2 & 2 \\ 0 & 2 & 1 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 4 & 0 & 3 \\ 2 & 1 & 3 & 4 \\ 1 & 2 & 0 & 3 \\ 4 & 3 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

Fig. 4.6.16 shows the explanation o f DNS algorithm with this example.

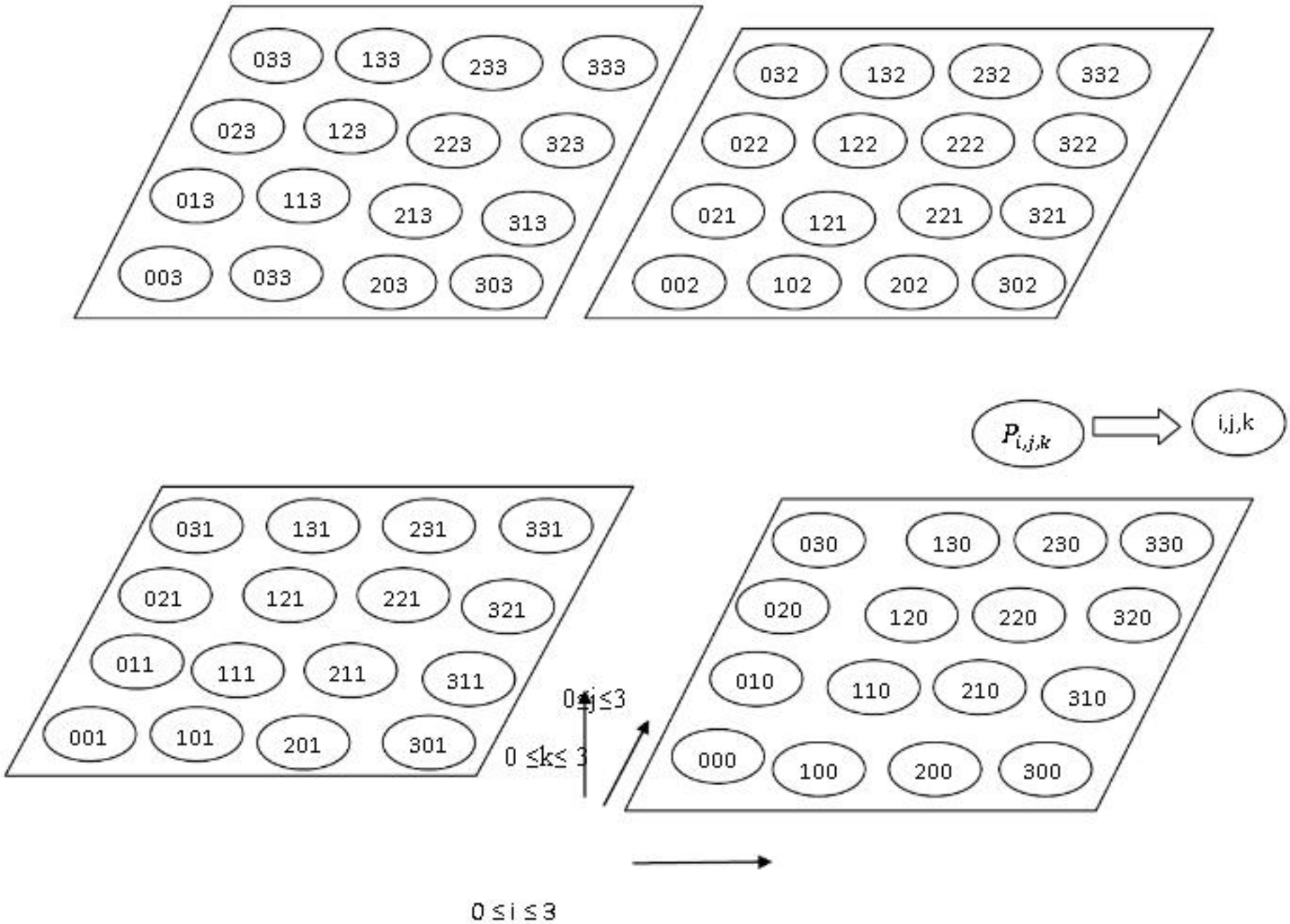
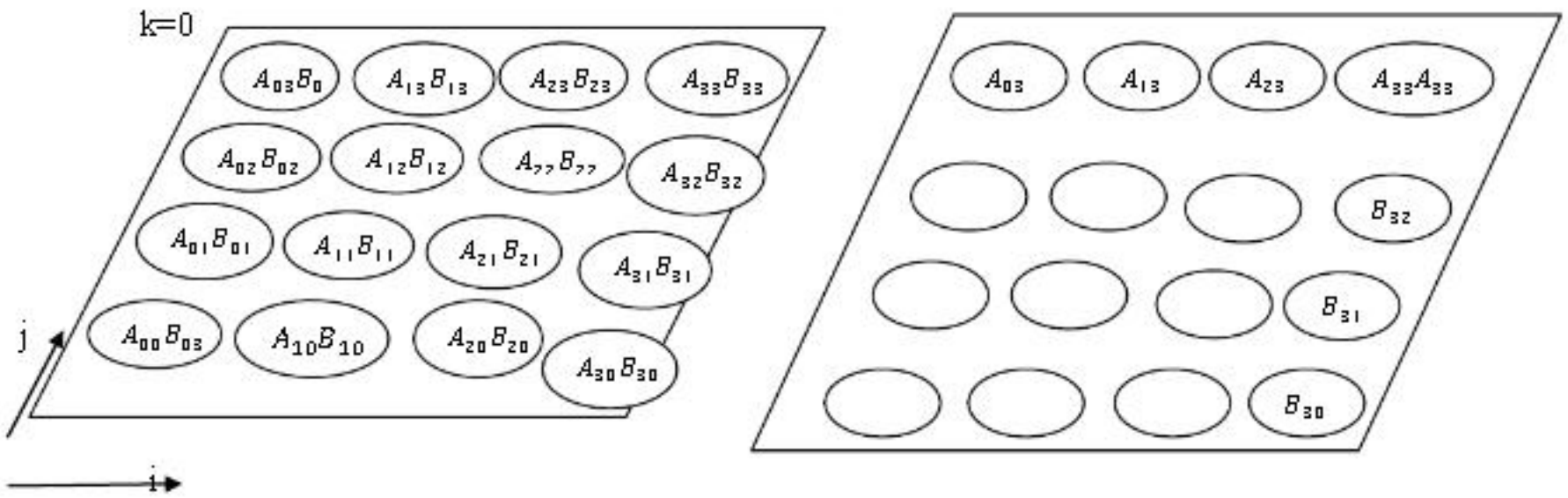
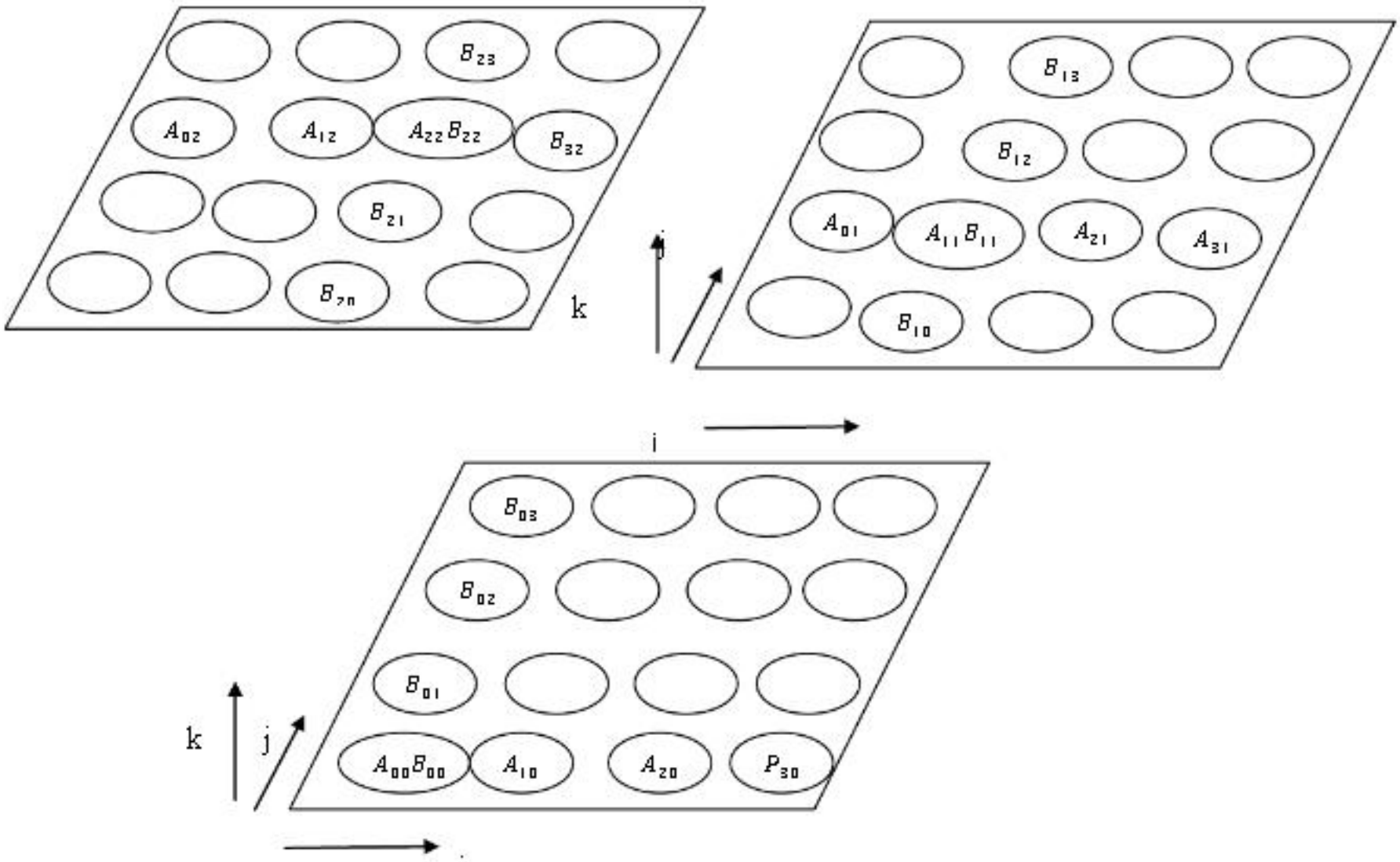


Fig. 4.6.13 : Logical array representation of n^3 processes

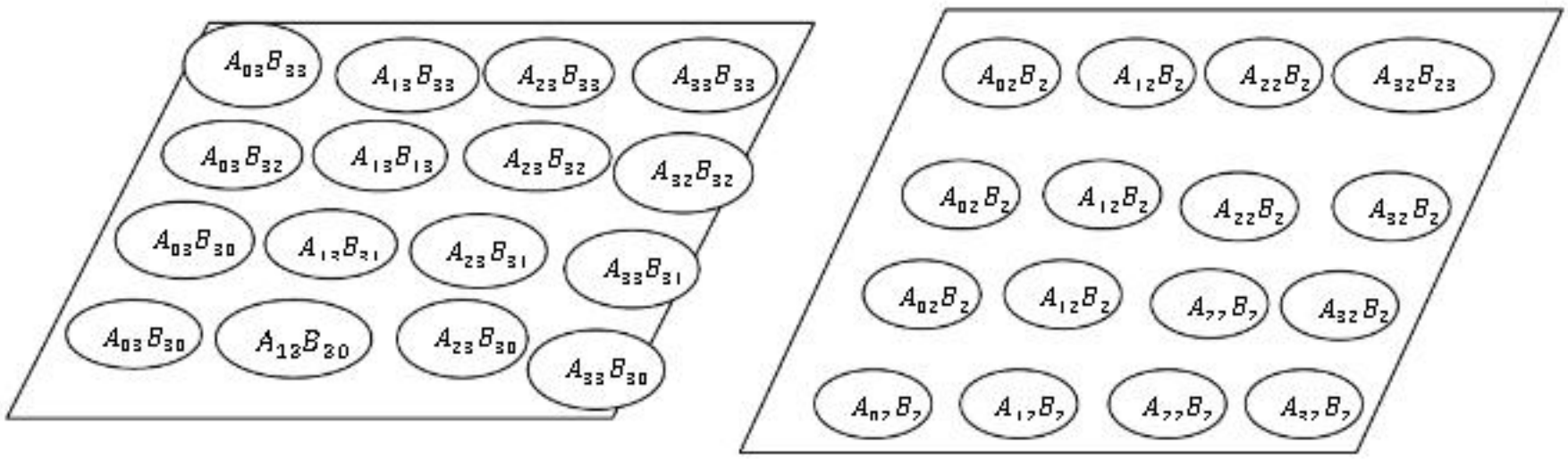


(a) Initial distribution of A and B matrices



1

(b) Moving jth row of A and ith column B to the plane, k=j



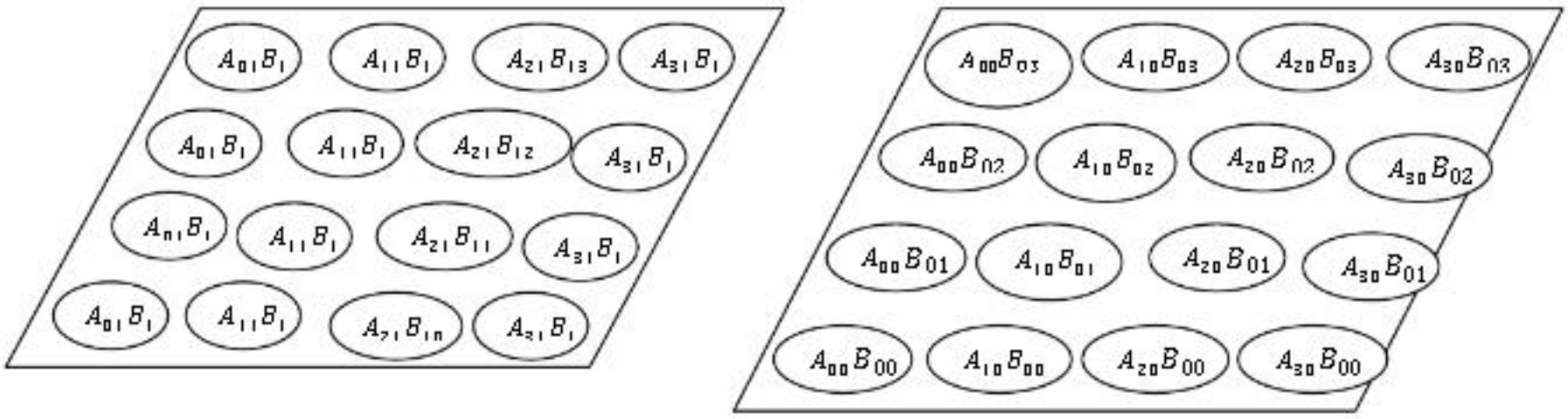
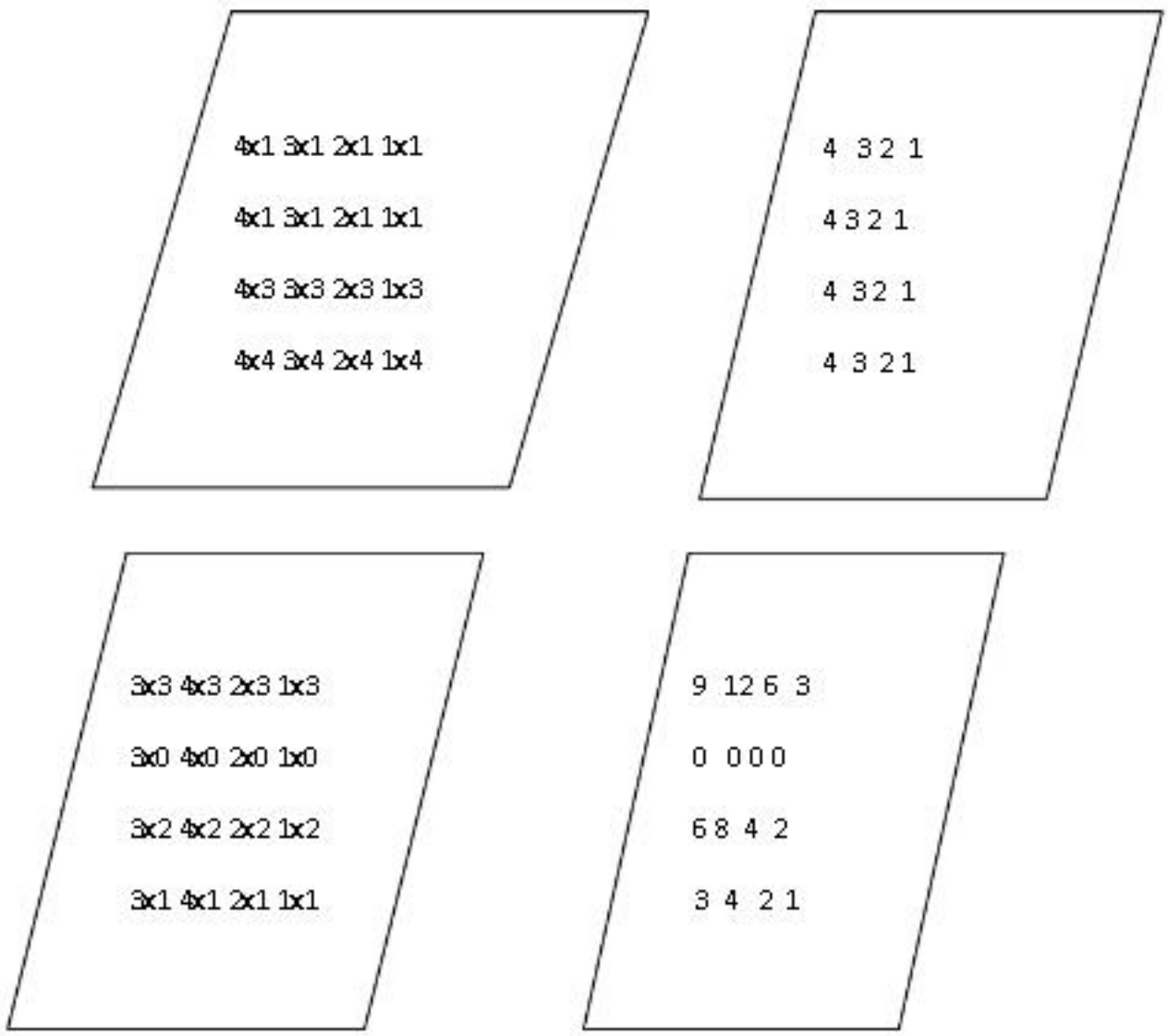


Fig. 4.6.15 Allocation of 2 elements to each processor after broadcasting



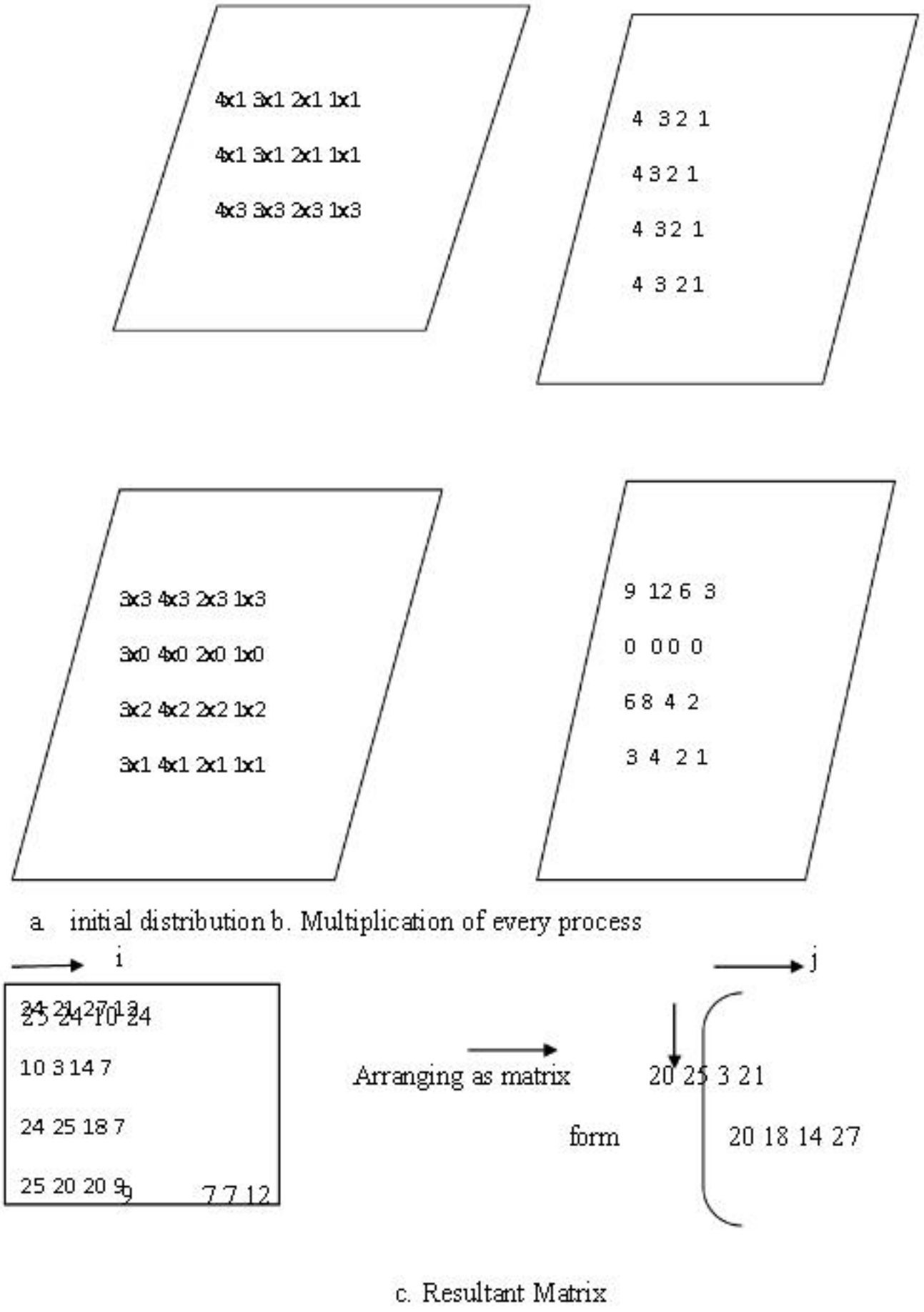


Fig. 4.1.16 Examples of matrix multiplication using DNS algorithm

UNIT- V Searching, Sorting and Graph Algorithms

5.1 Sorting

- One of the most commonly used operations of a computer is sorting. Many algorithms require the data to be stored, as they become easy to manipulate or search.
- Sorting is a process of arranging a collection of elements into either ascending or descending order. If $L = < x_1, x_2, ..., x_n >$ is the original sequence then after sorting or get $L` = < x`_1, x`_2, ....., x`_n >$ where $x`_i \leq x`_j$ and $1 \leq i \leq j \leq n$.
- There are number of sequential sorting algorithms like selection, bubble, insertion with complexity $O(n^2)$ and merge, quick, heap with complexity O(n log n).
- When sorting is to be done parallel, the algorithm must be designed must be designed by considering the parallel network to be used.
- The sorting algorithms can be classified as internal and external and comparison based or non comparison based. In internal sorting, the elements to be sorted or stored in the process's main memory, where as in external sorting, an auxiliary storage like tape or disk must be used. A comparison-based algorithm sorts a sequence by repeatedly comparing the elements and swapping them if they are not in the order. In a non comparison based algorithms, use other known properties of data like binary representation or distribution of data along with comparison the values. In this chapter, we discuss internal and comparison based algorithms .

5.1.1 Issues in Sorting on Parallel Computers

- When a sequential sorting algorithm is converted to parallel algorithm, the major task is to distribute the elements to be sorted onto the available processes. Following are the issues that are to be addressed while distributing.

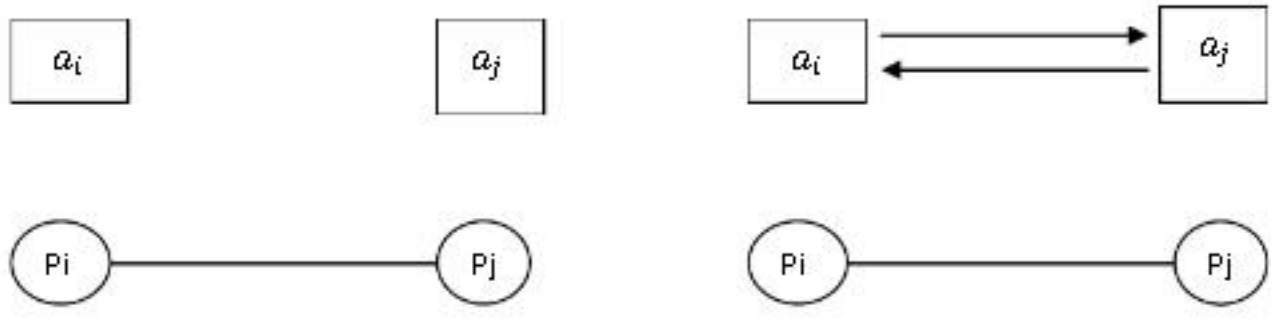
5.1.1.1 Where the input and output sequences are stored

- In a sequential algorithm, the processes memory always stores the input and output. But it is not necessary in case of parallel sorting. The data may be present either in only one process or may be distributed thought the processes. Wherever sorting is a part of any other algorithm, it is better to distribute the data among processes. We consider the same in this chapter.
- To distribute the sorted output sequence among the processes, the processes must be enumerated. This enumeration must be used to specify the order of data. Thus if the pocess pi comes before pj in the enumeration, then all the elements in pi occur before those stored

in pj in the sorted sequence. This enumeration depends on the type of interconnection network

5.1.1.2 How comparisons are performed

- As in case of sequential algorithms, the data to be stored is present in one process's memory, it is easy to perform compare- exchange operations. Whereas in parallel algorithm, the data is distributed among different processes. There can be two cases as discussed below.
- **Case 1:** Each process has only one element let a_i be present at pi and a_j at pj. Let the process pi comes before pj. To arrange a_i and a_j in ascending order, first thing is to send element from one process to another, and compare them in the process. Now pi keeps smaller value among a_i and a_j whereas pj keeps the larger. This is shown in Fig. 5.1.1



(a) Initial State **(b) Both processes send their data to other**

(c) Both processes have both data **(d) a_i and a_j are stored**

Fig. 5.1.1

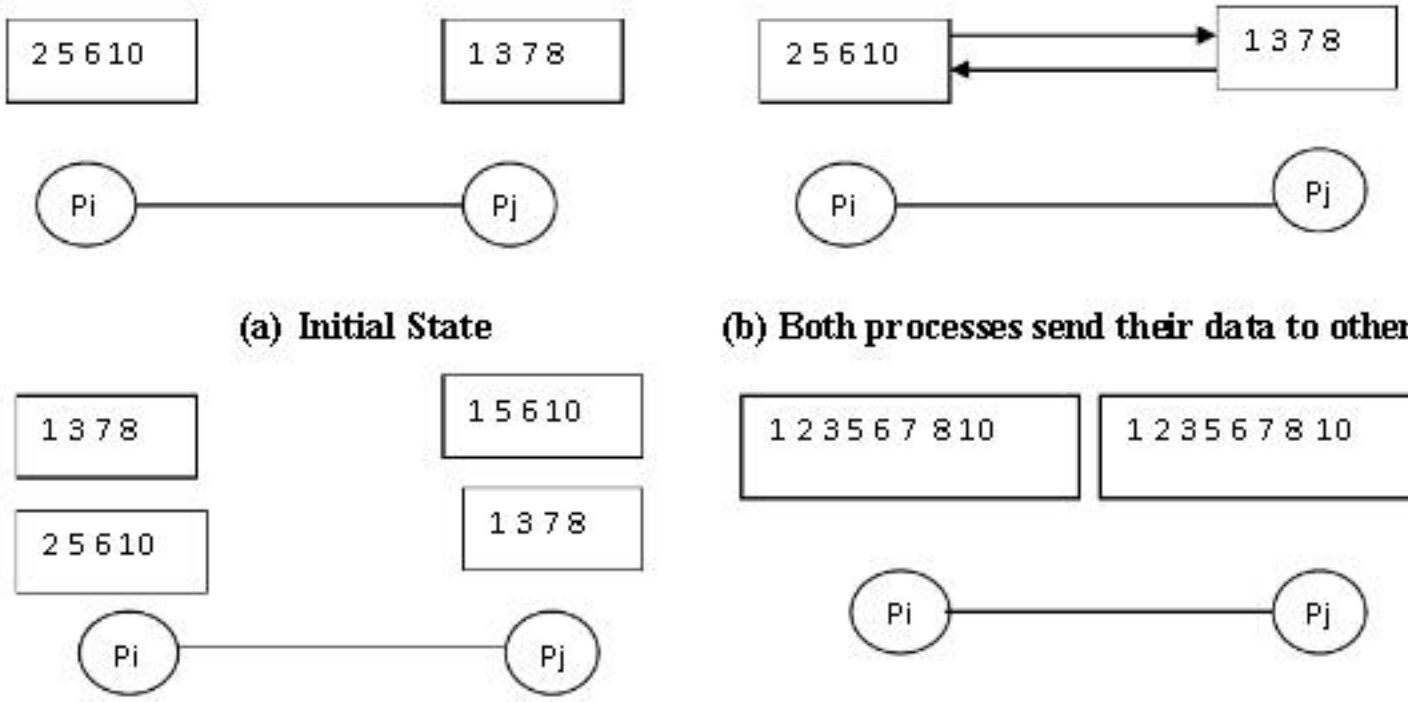
- If we assume that pi and pj are neighbors in the interconnection network (generally they will be) and there is bidirectional communication channel, then the communication cost is $t_s + t_w$. Practically, the startup time t_s is much greater than word transfer time t_w. In today's parallel computers, more time is required for inter process communication, than comparison.
- **Case 2 :** Each process has more than one element. When there is a large sequence to be sorted, a process stores more than one element in its memory. If there are n elements to

be stored, and p is the number of processes then, n/p elements are allotted to each process. The elements are divided into p blocks $A_0, A_1, \ldots, A_{p-1}$. There blocks are assigned to $P_0, P_1, \ldots, P_{p-1}$ respectively. If every element in A_i is less than or equal to every element in A_j then we say that $A_i < A_j$.

- After sorting the following will be true each process P_i has a set A` such that,

  $A` \leq A`$ where, $i \leq j$ and $\bigcup_{i=0}^{p-1} A_i = \bigcup_{i=0}^{p-1} A`$.

- In this case also, the two processes, send their data to other process. Here we assume that the elements in one process are already sorted. Then each process merges the data and half of the list is kept in the process.
- For example, in P_i, first half of the sequence is kept and in P_j, seconds half. This operation is called compare-split. This is shown in Fig. 5.1.2.
- Again by assuming that P_i and P_j are neighbors, and a bidirectional communication channel has the complexity $O(t_s + t_w\, n/p)$. As n/p increases, the initial cost t_s becomes negligible. Thus the cost becomes O(n/p).

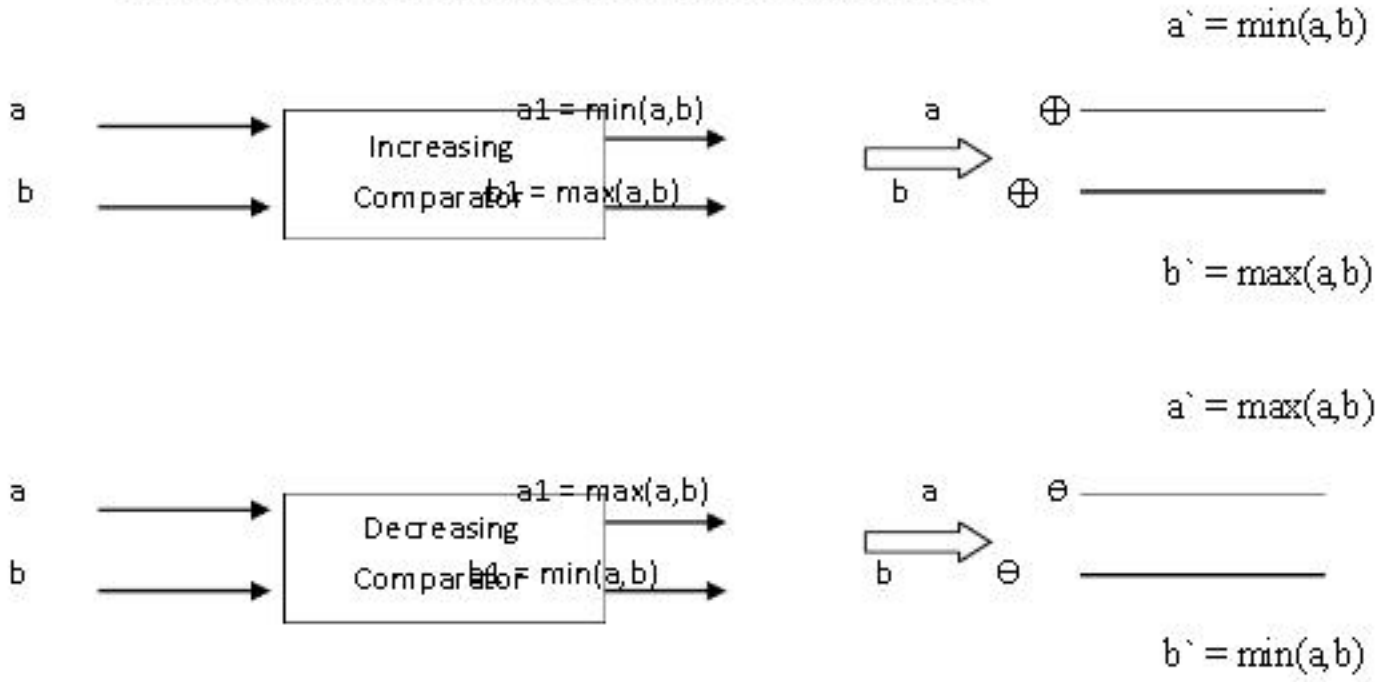


(e) First half kept in P_i and second half P_j

Fig 5.1.2

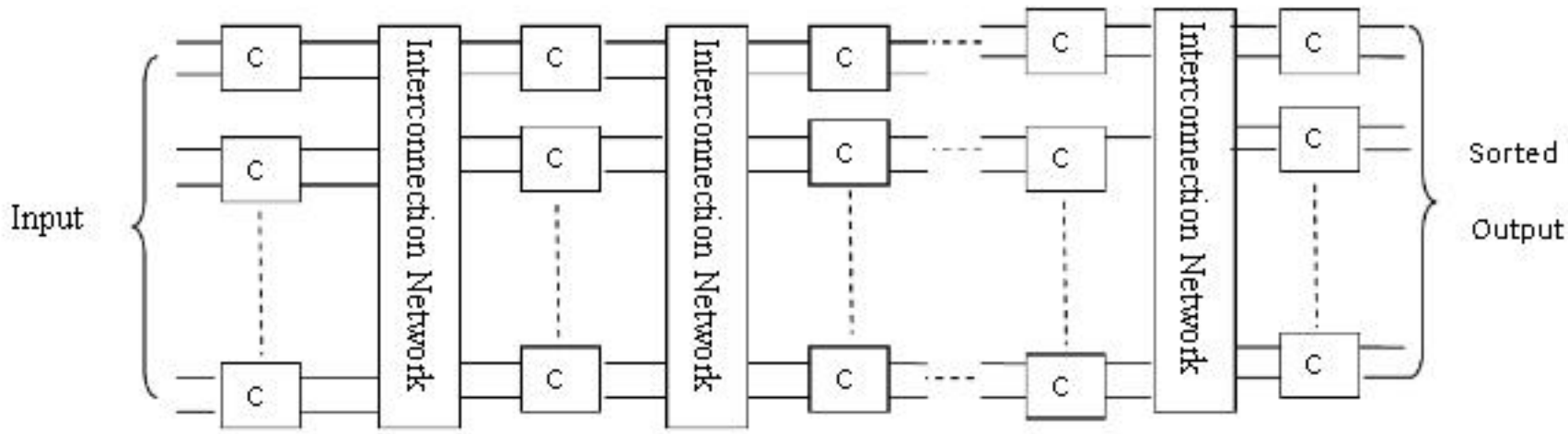
5.1.2 Sorting Networks

- The networks that we discuss in this section are based on a comparison network model, in which many comparisons are performed simultaneously.
- A component called a comparator, plays a major role in the network. It is a device which takes two inputs a and b, and generates two input a` and b`. There are two types of comparators : increasing and decreasing.
- In an increasing comparator, a` = min (a, b), b` = max (a, b), it is represented by ⊕.
- In a decreasing comparator, a` = max (a, b), b` = min (a, b), it is represented by ⊖.
- Fig 5.1.3 shows the representation of these comparators.



5.1.3 Representation of comparators

- In general, a sorting network is made up of series of columns. Each columns contains a number of comparisons, that are connected in parallel. The number of columns present in the network is called depth of the network.
- Each column performs a permutation and the stored output is taken from the last column. Fig 5.1.4 gives a schematic diagram of sorting network .



5.1.4 Schematic diagram of a sorting network

5.1.2.1 Bitonic Sort

- A sequence of elements $<a_0, a_1, \ldots\ldots, a_{n-1}>$ is said to be bitonic sequence, if
  i. There exists an index i, $0 \leq i \leq n-1$ such that the sequence $<a_0, a_1, \ldots\ldots, a_i>$ is in increasing order and the sequence $<a_{i+1}, a_{i+2}, \ldots\ldots, a_{n-1}>$ is in decreasing order. The other way may also be true i.e. decreasing sequence followed by increasing.
  ii. We can shift the sequence cyclically, so as to satisfy condition (i). For example, the sequence <1 5 7 9 6 3 2> is bitonic, because <1 5 7 9> is increasing order and <6 3 2> is in decreasing order. The sequence <5 7 6 4 1 0 2 3> is also bitonic, because we get the sequence <2 3 5 7 6 4 1 0> by cyclic shifting by 2 elements.
- A bitonic sort network sorts n elements in $O(log^2 n)$ time. The major operation in this network is to rearrange a bitonic sequence into sorted sequence. The first step is called bitonic split. This is done by the following procedure.
- Let, $s = <a_0, a_1, \ldots\ldots, a_{n-1}>$ be a bitonic sequence such that $a_0 \leq a_1 \leq a_2 \leq \ldots\ldots \leq a_{n/2-1}$ and $a_{n/2-1} \geq a_{\frac{n}{2}+1} \geq \ldots\ldots \geq a_{n-1}$.
- Consider the following sequences,
  $S_1 = <\min(a_0, a_{n/2}), \min(a_1, a_{\frac{n}{2}+1}), \ldots\ldots, \min(a_{\frac{n}{2}-1}, a_{n-1})>$
  $S_2 = <\max(a_0, a_{n/2}), \max(a_1, a_{\frac{n}{2}+1}), \ldots\ldots, \max(a_{\frac{n}{2}-1}, a_{n-1})>$
- There exists an element a_k in S_1 such that all elements before a_k are in ascending order and all elements after a_k are in descending order. There exists an element a_m in S_2 such that all elements before a_m are in descending order and all elements after a_m are in ascending order. Thus the sequences S_1 and S_2 are also bitonic.

- Fig 5.1.2.1 shows an example

$$S = <2\ 4\ 7\ 9\ 8\ 6\ 5\ 3>$$

$$S_1 = <2,4,5,3>$$

$$S_2 = <8,6,7,9>$$

Fig 5.1.5 Bitonic Split

- A bitonic sequence divided into two bitonic sequences. The bold numbers indicate change in order.
- The process of bitonic split can be recursively continued until there remains one element in the sequence. Now the output is in ascending order. After every bitonic split, the size of the sequence is reduces to half, the complexity is $O(log_n)$.
- The process of sorting a bitonic sequence using bitonic split is called bitonic merge. Fig 5.1.6 gives an example of converting bitonic sequence to sorted sequence. It is equally divided into two parts. For simplicity we consider n, as a power of two, in this case it is 16.

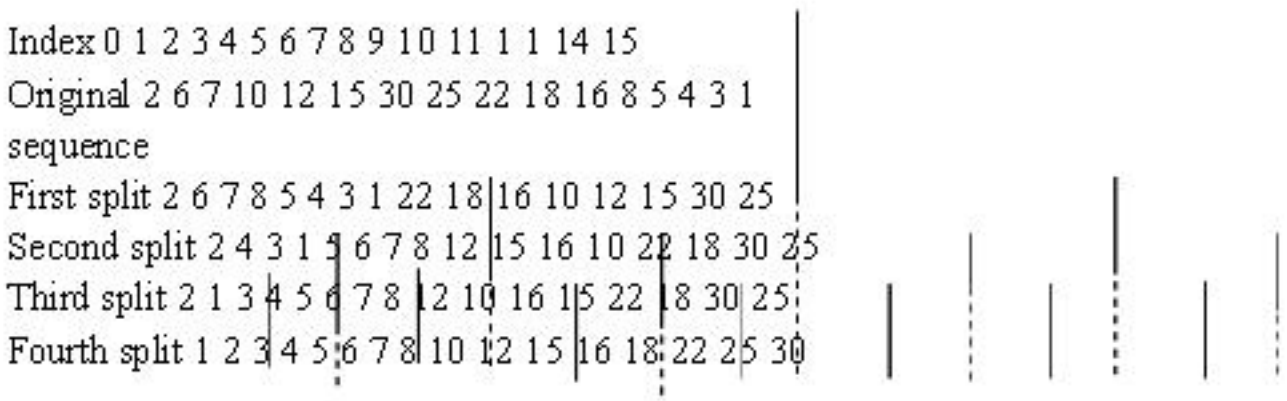


Fig 5.1.6 Sorting bitonic sequence

- The number of splits required is $log_2\ 16 = 4$. The network that is required for that process is called bitonic merging network. For n number in the sequence, the network has log n stages.
- Every state has 8 comparators; Let the index of input number can be represented by a binary number. The binary number represents the address of the input wire connected to the comparator. A comparator at stage k connects two number whose indices differ in kth most significant bit.
- Thus the number at index 3(0011) is connected to the number at index 11(1011) in stage 1, 7(0111) in stage 2, 1(0001) in stage 3 and 2(0010) in stage 4, in case of 4-stage network. Fig 5.1.7 shows an example.

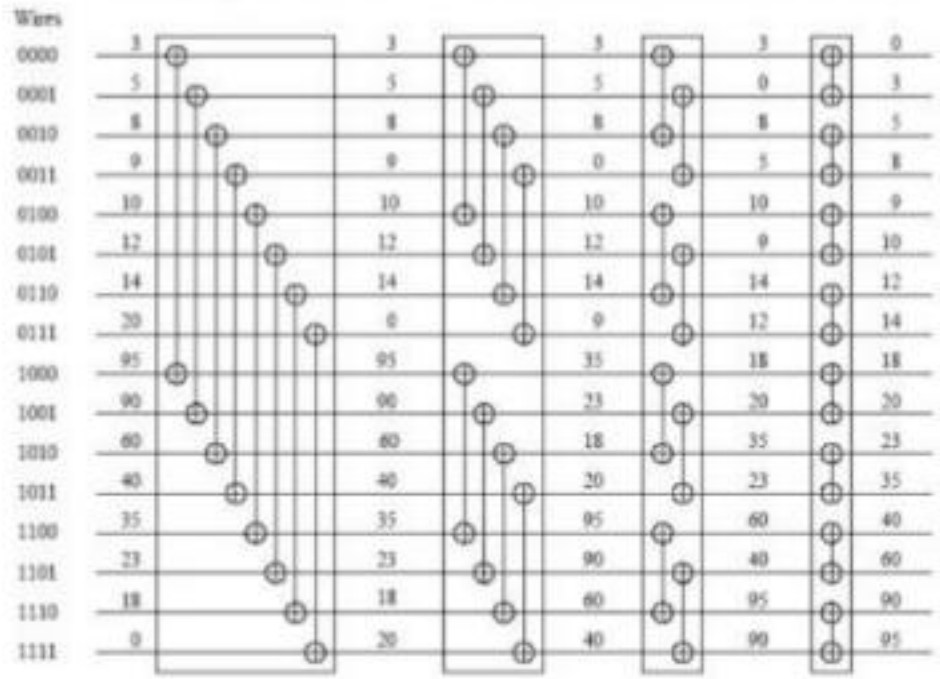


Fig. 5.1.7 Bitonic merging network example

- The sorting technique that uses bitonic merging network is called, bitonic sort. In this technique, an unordered sequence of elements is first converted to bitonic sequence and then sorted as shown in fig. 5.1.7.
- Let $\oplus$BM [k] represents bitonic merging network of k inputs that uses k/2$\oplus$ type of comparators, and $\ominus$ BM[k] represent bitonic merging network of k inputs that uses k/2 $\ominus$ type of comparators. The complete sorting procedure require log n stages, where the stage $log_{(n-1)}$ generates bitonic sequence. The stage 1 has n/2 bitonic merge networks arranged as alternate $\oplus$ BM[2] and $\ominus$ BM[2].
- Stage 2 has n/4 bitonic merge networks as alternate $\oplus$ BM[4] and $\ominus$ BM[4]. This continues till log nth stage, that has one bitonic merge network $\oplus$ BM[n].
- In general, a stage k has $n/2^k$ bitonic merge networks with alternate $\oplus$ BM[2^k] and $\ominus$ BM[2^k]. Fig. 5.1.8 shows the working of first three stages in details with an example.
- The third stage generates the bitonic sequence that we get in the bitonic sorting network is a perfect sequence. i.e. its first half is in increasing order and second half is in descending order.

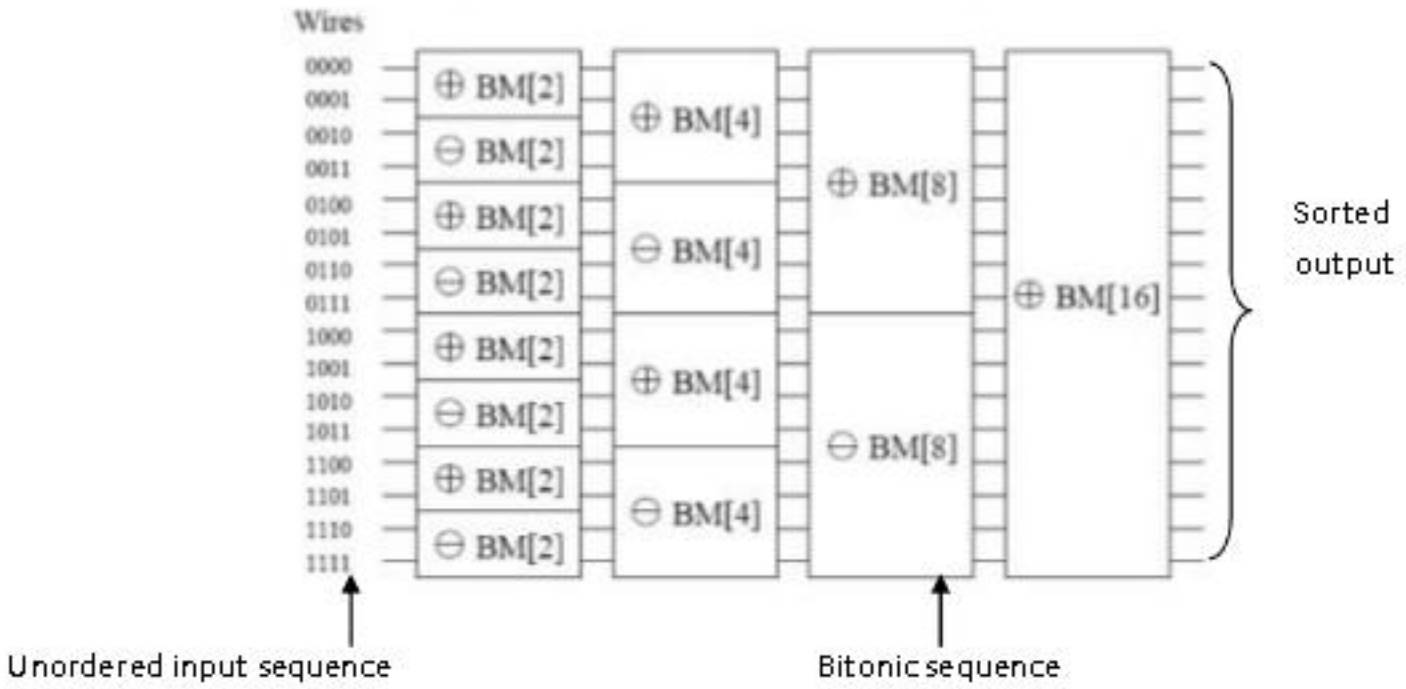


Fig. 5.1.8 Schematic diagram of bitonic sorting network

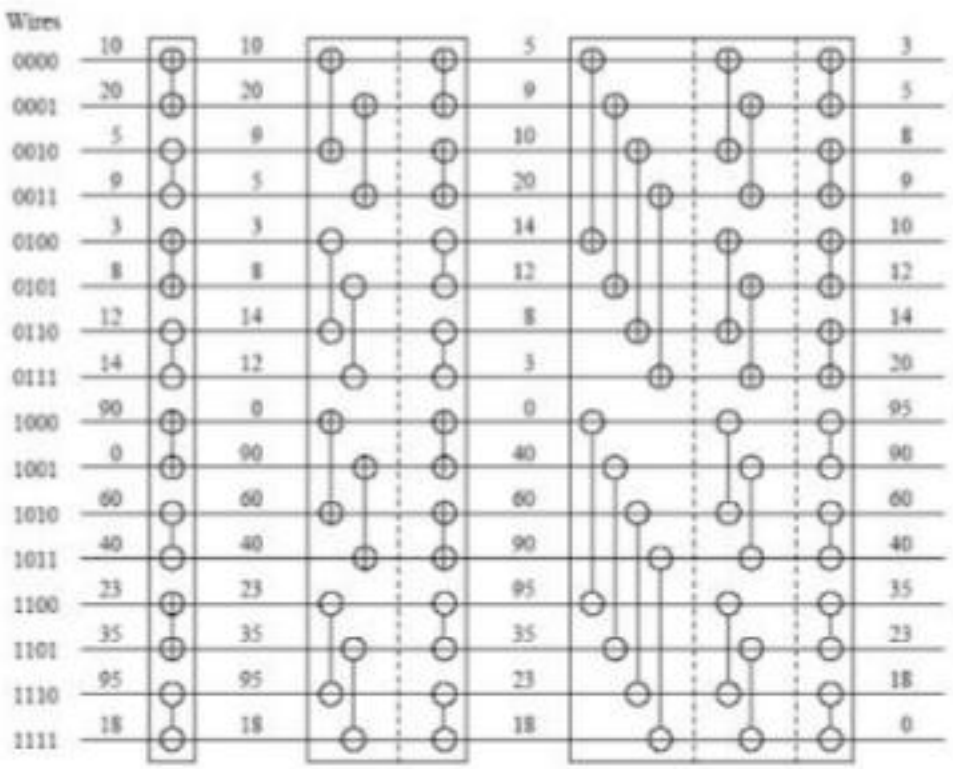


Fig. 5.1.9. Generation of bitonic sequence using first three stages of bitonic sorting network

- The bitonic sorting network has log n number of stages and the last stage has lo n substages. Thus the complexity of bitonic sort is given by, O(log^2 n).

5.2 Bubble Sort and its Variants

- Bubble sort is considered to be one of the simplest sorting techniques. The complexity of this algorithm is $O(n^2)$. There are many variants of bubble sort, which parallelize bubble sort and reduce the time complexity too.
- The sequential bubble sort algorithm takes the unordered input sequence $< a_0, a_1, \ldots\ldots, a_{n-1} >$. It has n-1 steps. In each steps, the a_i is compared with a_{i+1} where $0 \leq i \leq n-1$, and if $a_{i+1} > a_i$ then they are swapped. Fig. 5.2.1 shows a example. Following is the algorithm for bubble sort.

```
void bubble-sort(int a[],int n)
{
        for(i=n-2;i >=0 ;j--)
        for(j=0;j<i;j++)
        for(a[j] > a[j+1])
        swap(a[j], a[j+1]);
}
```

- Bubble sort, through is a simple algorithm, it can be easily parallelized may be with some modification. In the following sections, we discuss some of the variants of bubble sort.

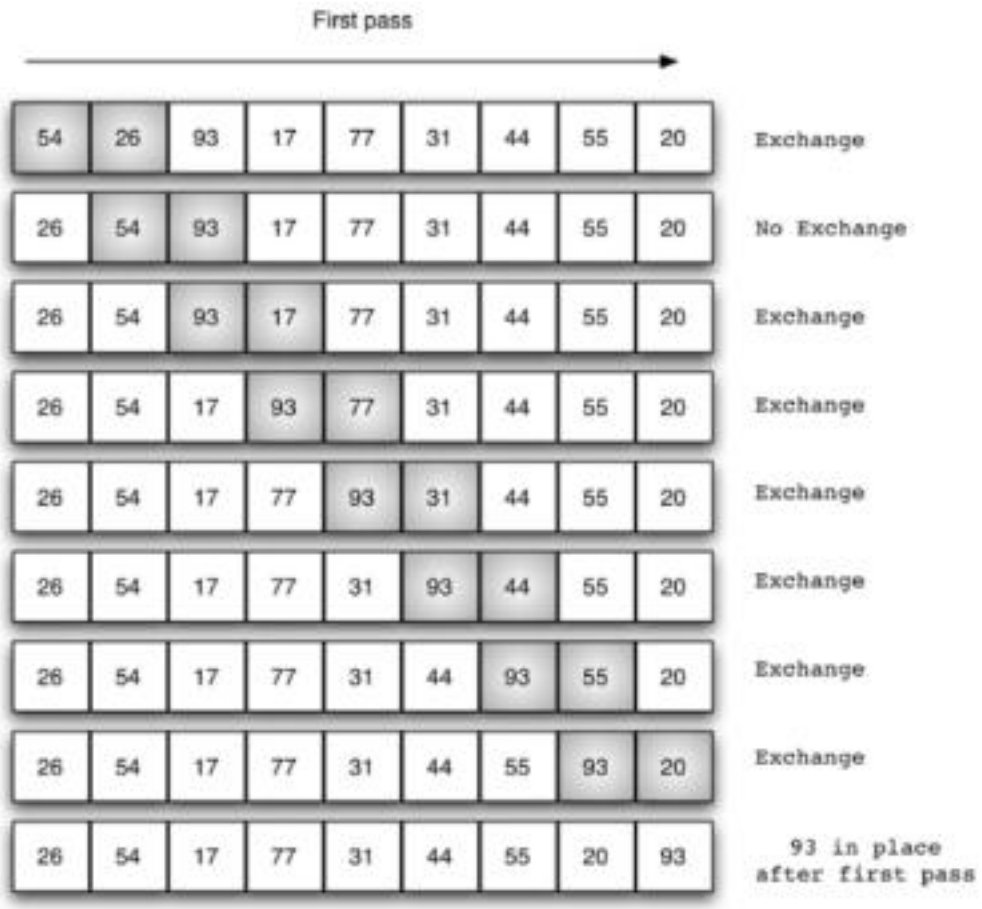


Fig 5.2.1 Example of Bubble Sort

5.2.1 Odd-Even Transposition Sort

- One of the simple variant of bubble sort is odd-even transposition sort. It has a n phases each phase, requires n/2 compare-exchange operation. In these n phases, there are alternated even and odd phases i.e. n/2 even phases and n/2 odd phase.
- In an even phase, every even indexed number is compared with its next number. If the first number if greater than the second, then they are swapped. Similarly in an odd indexed phase, every odd indexed number is compared with its next number and exchanged if necessary. Fig 5.2.2 gives an example.

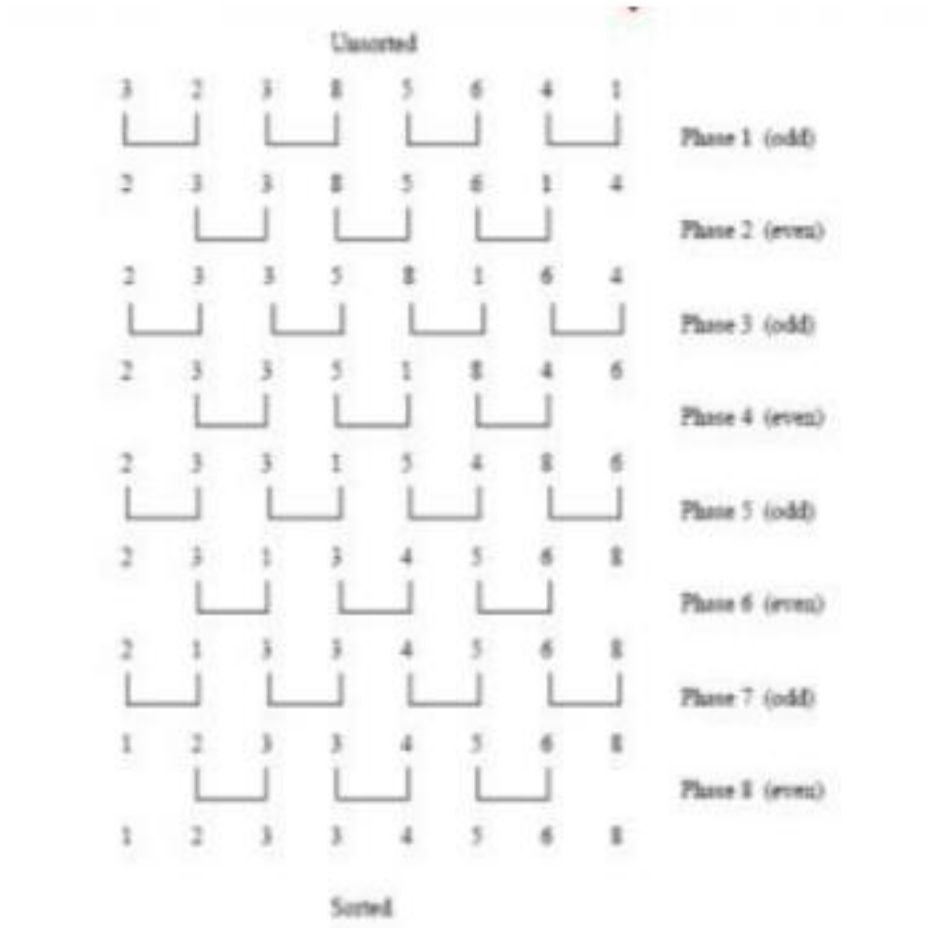


Fig. 5.2.2 Example of off-even transposition sort

- Following algorithm is used for odd-even transposition sort.

```
void odd_even(int a[],int n)

{

        for(i=1;i<=n;i++)

        {

                for(j=i/2;j<n-1;j=j+2)

                {

                        if(a[j]>a[j+1])

                        swap(a[j],a[j+1])

                }

        }

}
```

- When this algorithm us implemented as a sequential algorithm, the time complexity is $O(n^2)$, same as that of bubble sort. But this algorithm can be parallelized.
- The inner for loop compares the numbers at indices 0 and 1, 2 and 3 etc. for even phases and 1 and 2, 3 and 4 etc for odd phase. Thus successive iterations are independent of each other.
- Using this characteristic we can have n/2 processes, which compare 2 adjacent numbers and swap them if necessary. This operation taken O(1) time. The outer for loop has O(n) complexity. Thus n/2 process, the odd-even transposition has O(n) time complexity.

5.2.2 Shell Sort

- In odd-even transposition sort, the elements move only one position in every phase. If only few elements are out of order in the sequence, then also it takes same time when all elements are out of order.
- Shell sort is an algorithm that moves longer distance in a phase. Let n be the number of elements to be stored, and p be the number of processes. Let us assume that $p = 2^d$ for simplicity.
- A block of elements of size n/p is allocated to each process. The process are logically arranged as one dimensional array with indices from 0 to p-1. The shell sort algorithm has two phases.
- In the first phase, the elements in P_0 is compared with elements in P_{p-1}, and exchanged if necessary. Similarly P_1 and P_{p-2}, P_2 and P_{p-2}, P_2 and P_{p-2}, etc. are compared. In general the elements in process P_i, where $0 \le i \le n/2$ are compared with elements in P_{p-i-1}.
- Assuming that each process has single element, this is shown in Fig 5.2.3, continuing this in the next part, the process are divided into two parts and the same thing discussed above in performed. This is continued until there remains one process per group of processes.

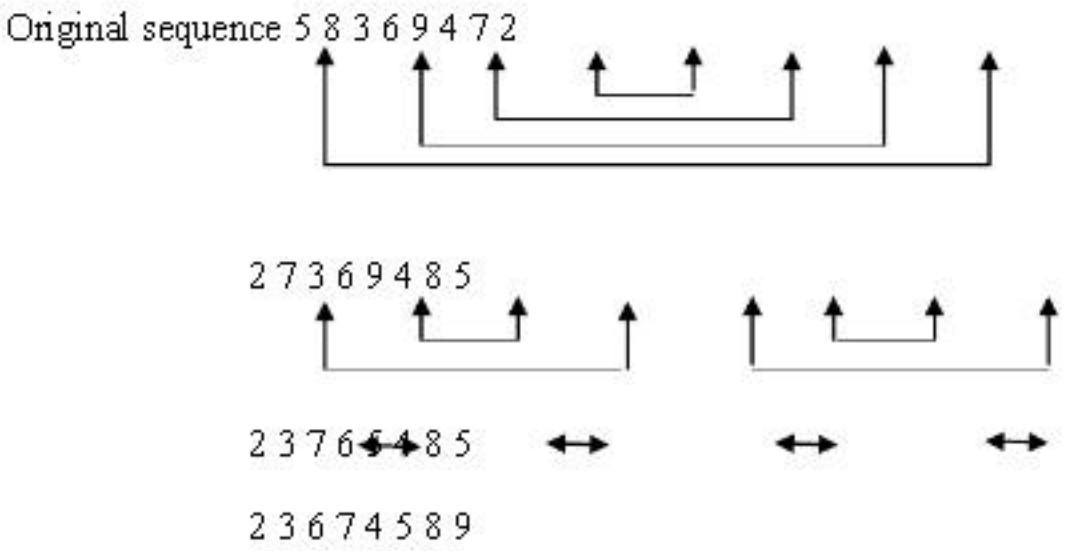


Fig. 5.2.3 First phase of shell sort example

- This is first phase of shell sort needs O(log n) time complexity. In the next phase, the usual odd-even transportation sort is performed. As the elements are moved nearer to their required places, this phase may not require O(n) time. Fig 5.2.3. shows the example of first phase of shell sort.
- In general, we can say that the worst case complexity of shell sort is O(n log n). The thing to be remembered is the second phase can be stopped, when there is no exchange operation.
- The best case complexity of shell sort algorithm would be O(log n) where the elements get sorted in the first phase. Fig 5.2.4 shows such one example. Readers can verify that the same sequence of numbers require 4 odd and 3 even phases if odd-even transposition sort is used.

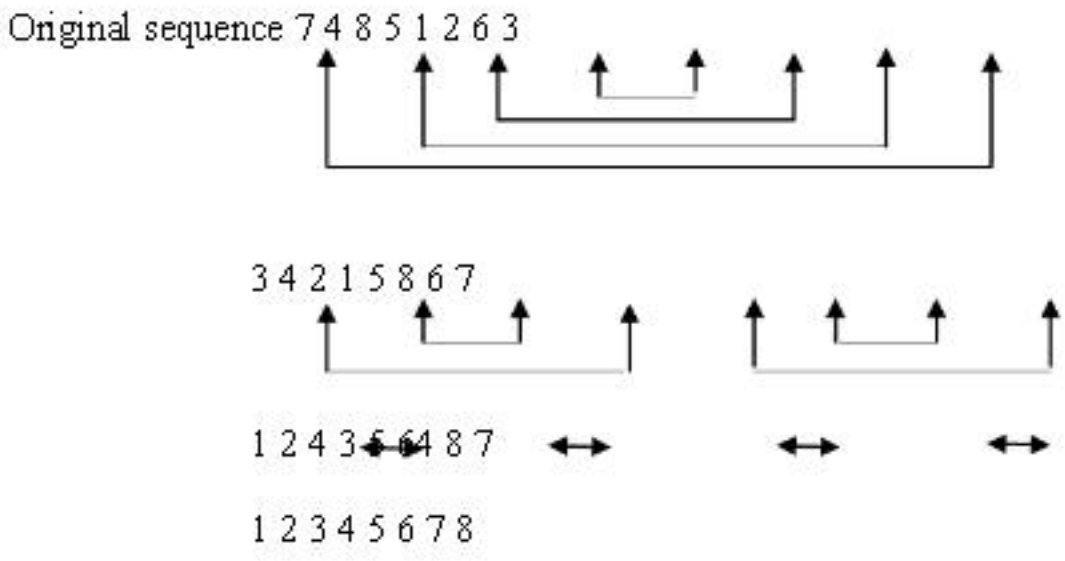


Fig. 5.2.4 Example in which shell sort has been time complexity O(log n)

5.2 Quick Sort

- Quick sort is one of the majority used sorting algorithm. It has the advantages like optional average complexity of O(n log n), low overhead and it is very simple in nature. It uses divide and conquer technique.
- The list of elements to be sorted is divided into two parts, and sorted separately. This division is done at an element k, such that all elements before k are less than or equal to k, and all elements after k are greater than equal to k. Thus after sorting the sub lists are separately, they need not be merged.
- To divide the list into two parts, we consider the first element in the list an pivot element. The remaining elements are compared with it. The pivot element is moved to its correct location, after the first phase. The quick sort algorithm is given below.

```
void quicksort(int A[i], int lb,int ub)

{

        if ub <= lb then

        return;

        x = A[lb];

        p=lb;

        for(i=lb+1;i<=ub;i++)

        {

                if(A[i]<=x)

                {

                        p=p+1;

                        swap(a[p],a[i]);

                }

        }

        swap(A[lb],A[p]);

        quicksort(A,lb,p);

        quicksort(A,p+1,ub);

}
```

- Fig. 5.2.1 shows working of first phase of quick sort in detail.

Index i	Array	P	Comparison	
0	~~8~~ 3	~~8~~		
1	~~8~~ 4		8≤5x	
2	~~4~~ ~~8~~ 2	1	4≤5√	Swap(8,4)
3	~~10~~ ~~8~~ 5		10≤5x	
4	~~2~~ 6	2	2≤5√	Swap(8,2)
5	~~8~~ 10	3	3≤5√	Swap(10,3)
6	9		9≤5x	
7	6		6≤5x	

End of for swap(5,3)

lb=0,ub=7,x=5

Array after first phase – (3 4 2) 5 (8 10 9 6)

Fig. 5.2.1 First phase explanation of quick sort

- The complexity of dividing the list into two parts takes O(n) time complexity. The list is divided into two parts after each phase, thus the complexity of quick sort is considered to be O(n log n).
- If the list an ordered list, then the list is divided into two parts, such that one list has only one element and other has all remaining elements. If this is the case, then quicksort algorithm will be recursively called n times, which makes its complexity to be $O(n^2)$.

5.2.1 Parallelizing Quick Sort

- There are many ways in which quick sort can be parallelized. One of the simple ways is to start with a single process. After finding the correct position for the pivot elements, create two processes for sorting the two sub lists.
- In the next phase, create 4 processes. This is continued till the list is sorted. Though this method seems to be simple, it uses only one process for partition i.e. to find the correct location for the pivot element. Thus the overall complexity still remains to be O(n log n).

5.2.2 Parallel Formulation for a CRCW PRAM

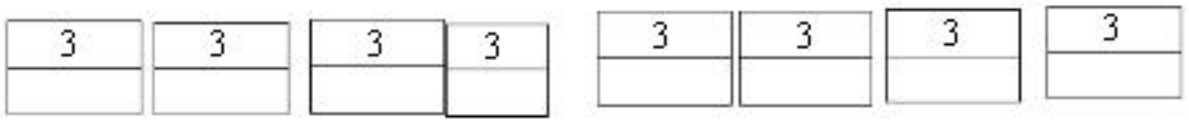
- CRCW PRAM is Concurrent Read, Concurrent Write, Parallel Random Access Machine. In this type of machine, when two or more processes, try to write into same memory location, then only one process is chosen and remaining are ignored.
- In this method, a binary search tree is created for the given sequence of numbers algorithm gives the construction of tree. The shared memory holds the variable root.
- All processors try to write it at the same time, but only one succeeds and exists. Similarly lc and rc for any processes are shared and only one process will write into them.

```
build-tree(int A[]int n) {
        for each process; {
                root = i;
                Parent[i] = root;
                lc[i] = rc[i] = n+1;
        }
        repeat for each process i ≠ root {
                if (A[i] < A[Parent[i]]) or
                (A[i] == A[Parent[i]] an i < Parent[i]) {
                        lc[Parent[i]] = i;
                        if(i = lc[parent[i]])
                        re turn;
                        Parent[i] = lc[Parent[i]];
                }
                else {
                {
                        rc[parent[i]] = i;
                        if(i = rc[Parent[i]])
                        return;
                        Parent[i] = rc[Parent[i]];
                }
        }
}
```

- Fig. 5.2.2 explains this method. The average run time of this method is O(log n). The sequence of processes that write into shared memory is.

Step 1 :

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Array	**5**	**3**	**8**	**4**	**6**	**10**	**2**	**7**
	P0	P1	P2	P3	P4	P5	P6	P7

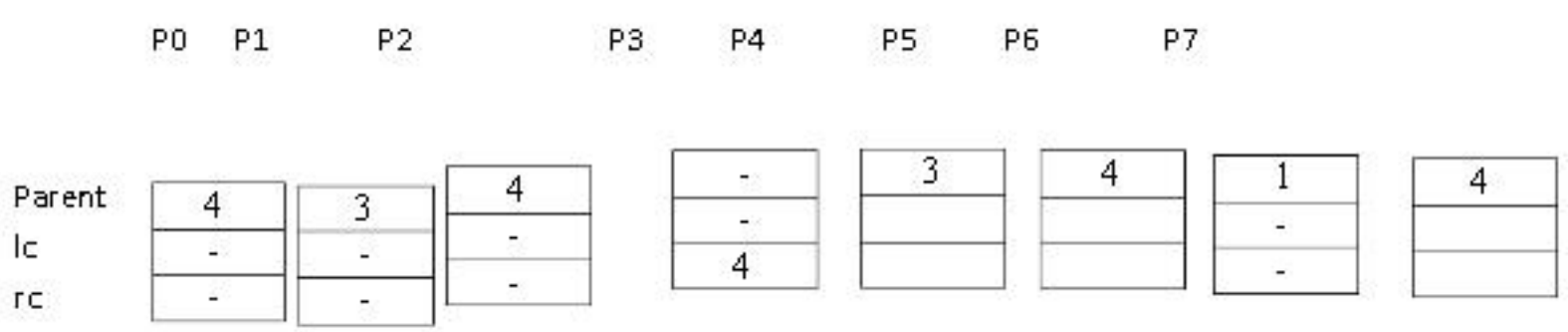


Step 2 : A[1], A[6] < A[Parent[i]] (it is 4 in this case)

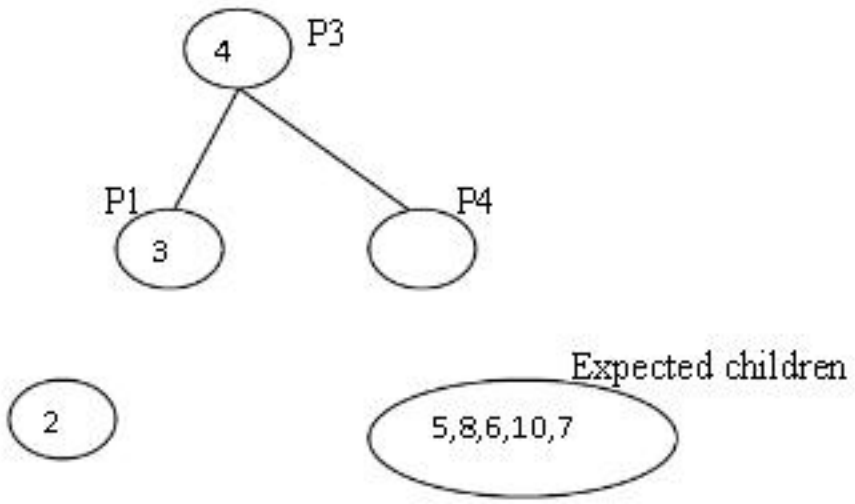
Let P1 writes and exit

A[0], A[2], A[4], A[5], A[7] > A[Parent[i]]

Let P4 write and exit.



Equivalent tree



Step 3 : A[2], A[5], A[7] > A[Parent [i]] {it is 6}

A[6] > A[Parent [i]]

Let P0, P2 write and exit

A[6] > A[Parent [i]] {it is 3}

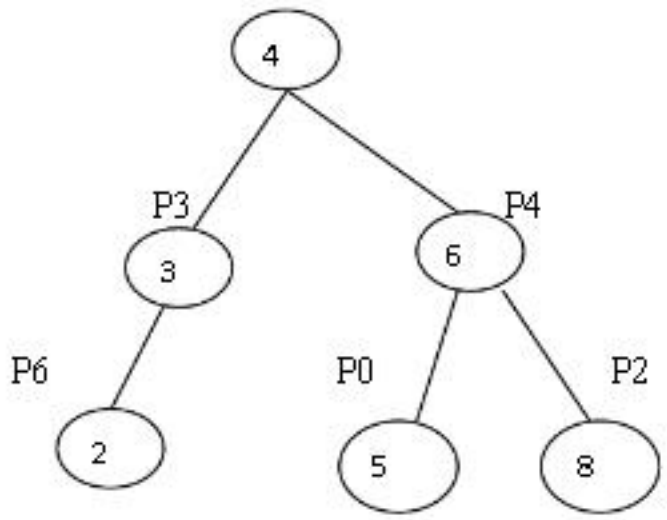
P6 writes and exists

Let P4 write and exit

P0 P1 P2 P3 P4 P5 P6 P7

Parent	4	3	4	-	3	2	1	2
lc	-	6	-	1	0		-	
rc	-	-	-	4	2		-	

Equivalent tree P1



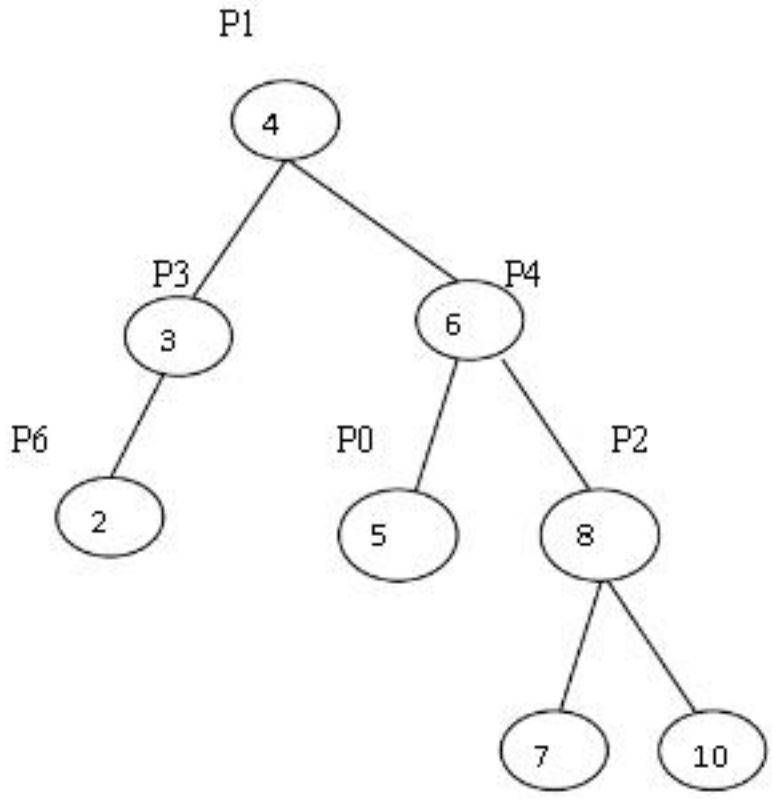
Step 4 : A[5] > A[Parent [i]] {it is 8}

A[7] < A[Parent [i]] {it is 8}

P0 P1 P2 P3 P4 P5 P6 P7

Parent	4	3	4	-	3	2	1	2
lc	-	6	-	1	0		-	
rc	-	-	-	4	2		-	

Equivalent tree P1



Indicates process that exists

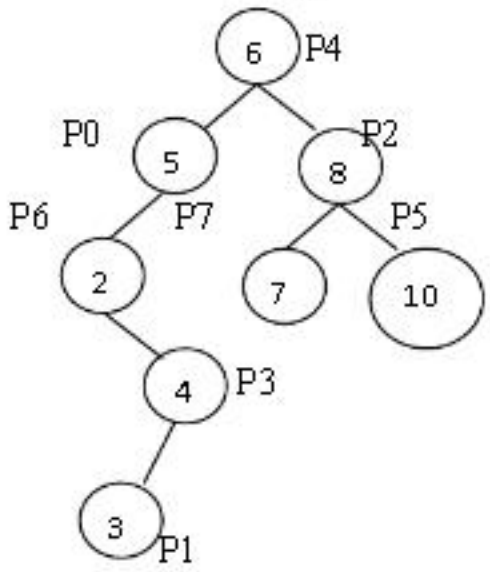
Fig 5.2.2 Working of parallel quick sort algorithm on CRCW PRAM

P3, P1, P4, P0, P2, P5, P7. If this sequence is changed, we get a difference tree. Fig 5.2.3 shows different trees constructed with different sequence of processes.

Sequence of process that exit

P4, P0, P2, P6, P7, P5, P3, P1

Equivalent tree



P2, P3, P5, P1, P0, P7, P6, P4

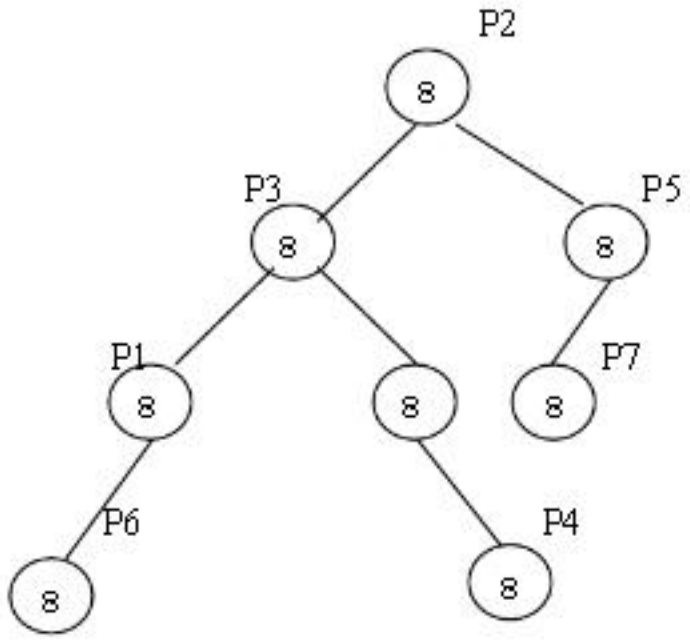
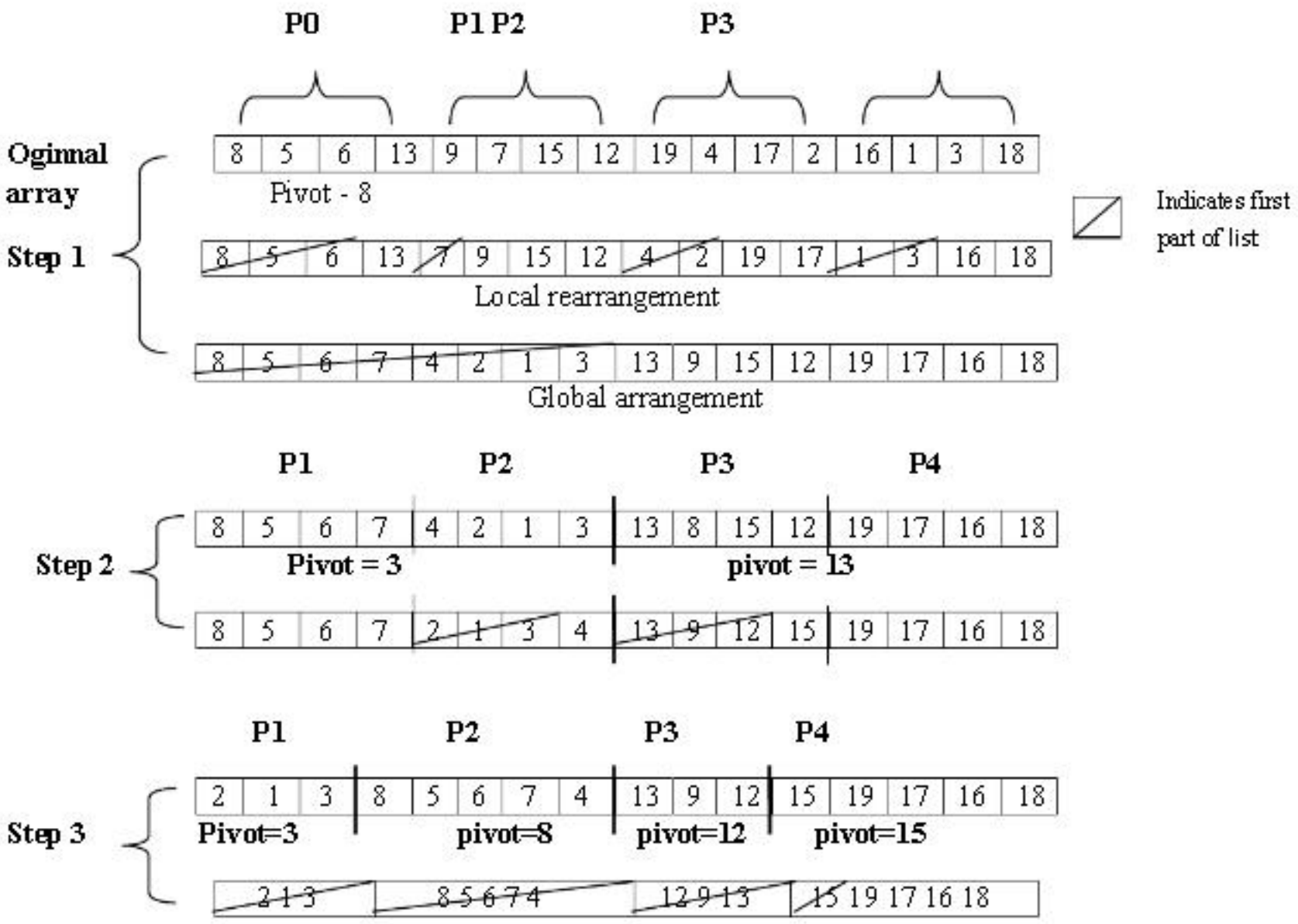


Fig. 5.2.3 Examples of trees constructed for different sequence of processes



Step 4

1 2 3 4 5 6 7 8 9 12 13 15 16 17 18 19

sorted array

Fig. 5.2.4 Example of shared space parallel architecture for quick sort

5.2.3 Parallel Formulation for Parallel Architecture.

Now, let us consider a parallel architecture with p processors connected through an interconnection network.

5.2.3.1 Shared address space parallel formulation

- Let the processes; and n be the size of the array, then each process is allotted n/p elements in the array. Thus we can say that Ai is the block of array Ai, that is allotted to process Pi, where 0 <= i <= p-1.
- The first step is to select the pivot element which is generally the first element in the array and broadcasting it to all processes.
- When a process receives the pivot element, it partitions its allocated block into two parts such that the first part has all the elements that are less than or equal to the pivot element, and second part has all the elements that are greater than pivot element.
- This is done by each process using the partition technique discussed earlier. The next step of the algorithm is to rearrange the array so that the elements less than pivot from all processes are followed by pivot and then by elements greater then pivot.
- Now, the processes are partitioned into two groups, and sort them separately by recursively calling quicksort algorithm. Fig. 5.2.4 shows an example.

5.2.4 Pivot selection : A Crucial Task

- Quicksort is a divide and conquer algorithm which relies on a partition operation : to partition array, we choose an element, called a pivot, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in the linear time and in-place. The next procedure is to recursively sort the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.
- The worst case complexity of quick sort algorithm is $O(n^2)$, when the list is in a particular order and the leftmost or rightmost element is selected as pivot. However, this can be reduced with the correct choose of the pivot. A better way is to choose the median value from the first, the last, and the middle element of the array. One can go for a random selection of pivot as well.
- If the range of the numbers in the list of elements is too varying, then the median method does not help in reducing the complexity. In such cases, median of 5 or more numbers can be used.

- If the pivot selection is optimal, then only the complexity of the quick sort would be O(n log n).

5.3 Bucket and Sample Sort

5.3.1 Bucket Sort

- When the fact that in an array, the number are evenly distributed in a range [a,b], then we use an algorithm called bucket sort algorithm. Being evenly distributed means, in an interval [a,b] the array has elements.
- For example, [7,5,8,2,6,10] is evenly distributed where as [2,10,9,3,1,8] is not because the elements are concentrated only at the borders (1,2,3 and 8,9,10).
- The elements in the array are equally divided into P parts and assigned to P process. The intervals are also divided and assigned to the processes. The parts of intervals are called buckets.
- In each phase, the elements area stored into respective buckets. The contents of buckets are concatenated and the procedure is repeated. Fig 5.3.1 shows an example.
- Parallelizing bucket sort is a straight forward process. Let P be the number of processes equal to the number of buckets. Each process partitions its allocated block of n/p elements into P sub blocks.
- Then every process sends all sub blocks to respective processes. Then any optimal sequential algorithm can be used to sort the elements in every process.

5.3.2 Sample Sort

- The disadvantage of bucket sort is its basic assumption of uniform distribution of elements. This may not be possible in many cases.
- In such cases

Array 78 63 91 42 39 5 24 56 17 80

Buckets 0 1 2 3 4 5 6 7 8 9

As per unit place distribute elements 80 91 42 63 24 5 56 17 78 39

As per tenth place distribute elements 5 17 24 39 42 56 63 78 80 91

Sorted list

Fig. 5.3.1 Example of bucket sort

- Bucket sort is not efficient. A sorting technique called sample sort can be used in such cases. This is very simple algorithm that works similar to the bucket sort algorithm
- Given a sequence of elements, a sample space is selected and sorted. This sorted list of samples define the range of buckets. These elements are called splitters.
- They divided the given elements into buckets. The next procedure is same as that of bucket sort. Fig. 5.3.2 shows an example.

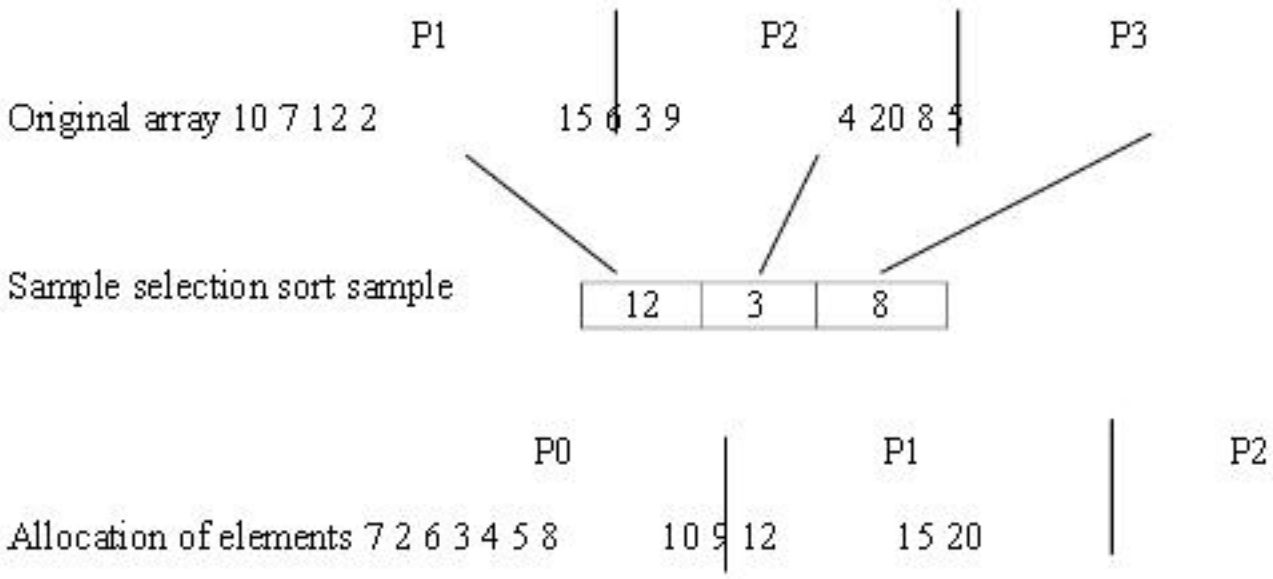


Fig. 5.3.2 Example of sample sort

5.4 Parallel Depth-First Search

- To solve a discrete optimization problem, depth first search algorithm is used if it can be formulated as tree search problem. The DFS expands a most recently generated node, and starts with the initial node by generating its successors.
- If any node has no successors, then it indicates that there is no solution in that path. Thus backtracking is done and continued to expand another node. Fig. 5.4.1 gives the DFS expansion of the 8-puzzle discussed earlier.
- The initial configuration is given in (a). There are only two possible moves, Blank up and blank right. Thus from (a) two children or successors are generated, those are (b) and (c).
- This is done in step 1. In step 2, any one of (b) and (c) is selected. If (b) is selected then its successors (d), (e) and (f) are generated. If (c) is selected then its successors (g), (h) and (i) are generated.
- Assuming that (b) is selected, and then in the next step (d) is chosen, it is clear that (d) is same as (a), thus backtracking is necessary. This process is continued until the required result is found.

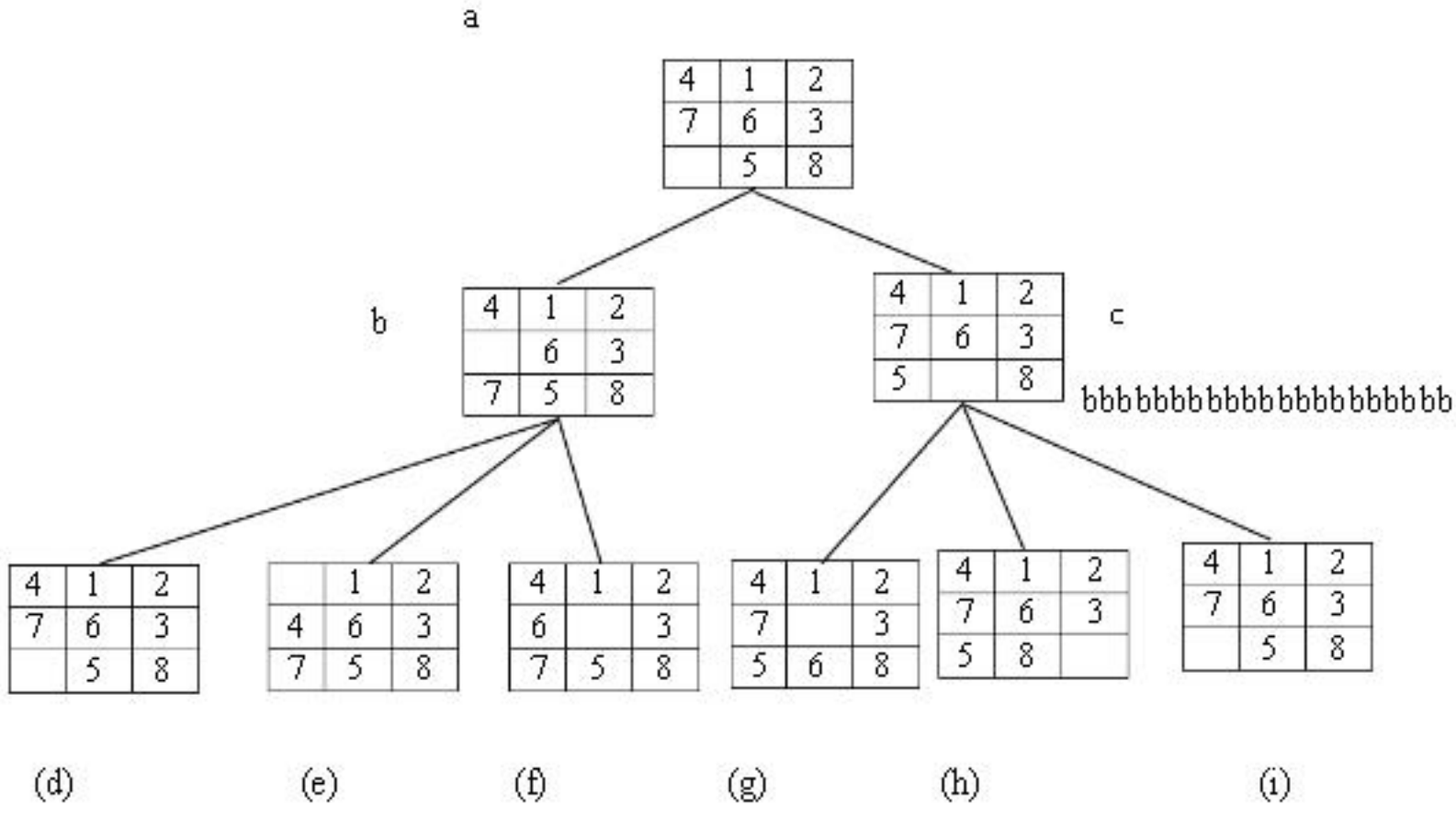
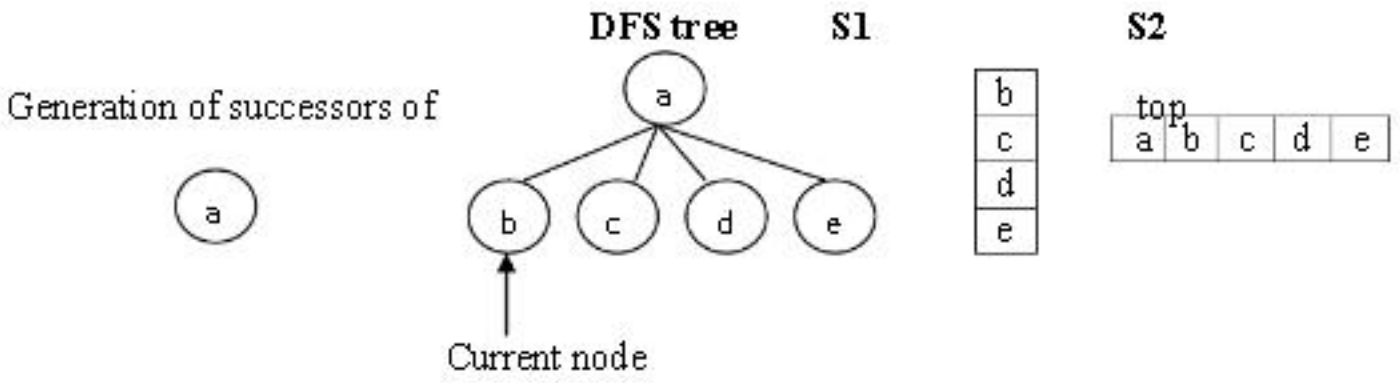


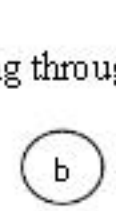
Fig. 5.4.1 Solving 8-puzzle using DFS

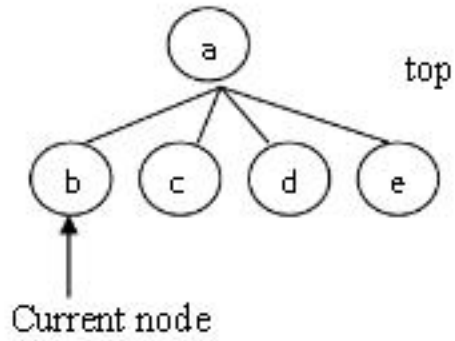
- Thus for each step, it is necessary to store the unexplored nodes. The information about the generated tree is store by using a stack.
- Fig. 5.4.2 show3s some examples of DFS trees along with two stacks S1 and S2. The stack S1 stores the information of unexplored nodes. The stack S2 has a different structure at each element.
- It consists of atleast one node which is parent and it is followed by its unexplored children.
- The dotted lines in the tree indicate the path which have already traversed. The nodes are pushed into the stack from right to left, so that the traversal is done from left to right.

DFS Tree :



Traversing through



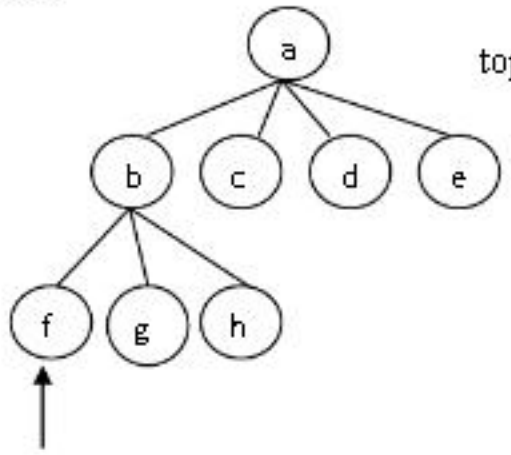


top

c
d
e

a	B	c	d

Generation of successors of



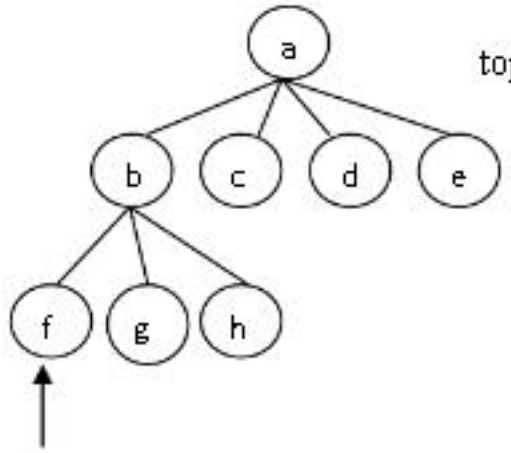
top

f
g
h
c
d
e

b	f	g	h
a	b	c	d

Current node

Traversing through



top

g
h
c
d
e

b	g	h	
a	b	c	d

Current node

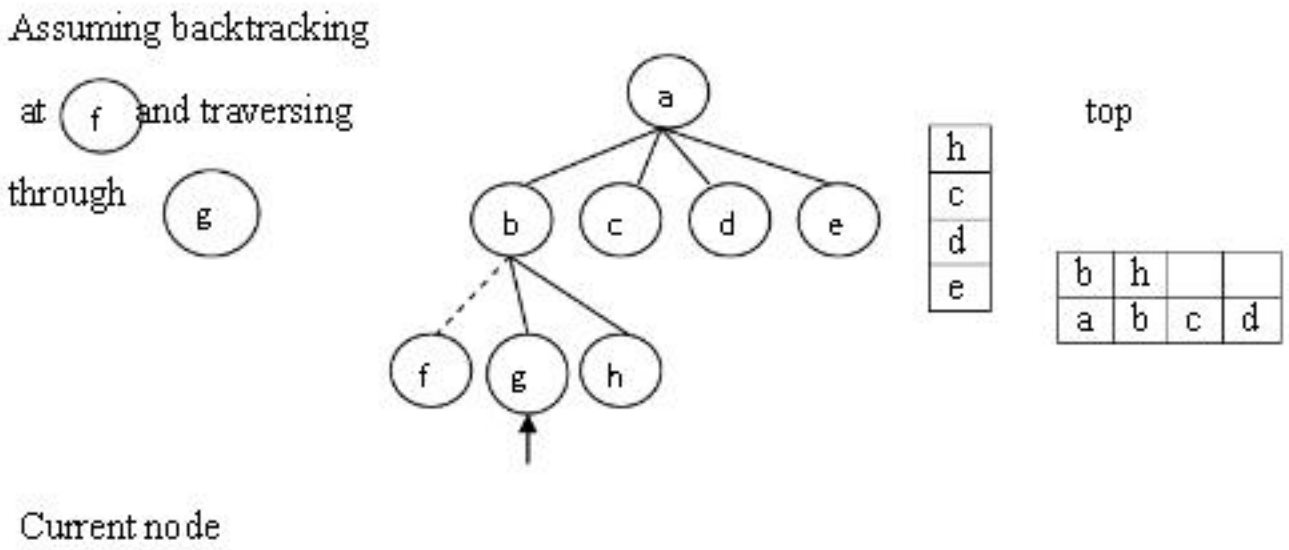


Fig. 5.4.2 Example of DFS trees and corresponding stacks

- Consider Fig. 5.4.3. It has a DFS tree that is explored upto 7 levels. The nodes b, f, g, i, , k, n and r are explored and backtracked. The current nodes is u and the current path of generation is a, c, j, l, o, s, t, u.
- The stack S1 has v on its top, indicating that, if backtracking is to e performed after u, then the next nodes to be expanded is v. The top of S2 is,

t	v	w

- It indicates that t is the parent of the current node and t has two unexplored successors v and w. When we convert a sequential DFS algorithm to a parallel DFS algorithm, the major issue that should be taken care is distributing the search space among the processors. There are two types of distribution, static and dynamic.
- In static distribution of processors, the sub trees of the dfs tree which are at the same level, are assigned to available processors.
- Consider the tree shown in Fig. 5.4.4. Initial expansion of root node A is done on one processor. If there are two processors under consideration, then one processor will expand the node B and other is responsible for C.
- This is shown in Fig. 5.4.5. If we consider 4 processors, then while expanding the nodes B and C, each of them are assigned to two processors while other two processors are kept idle. In the next step, the sub trees roots are D, E, F and G are assigned to four processors. This is shown in Fig. 5.4.6

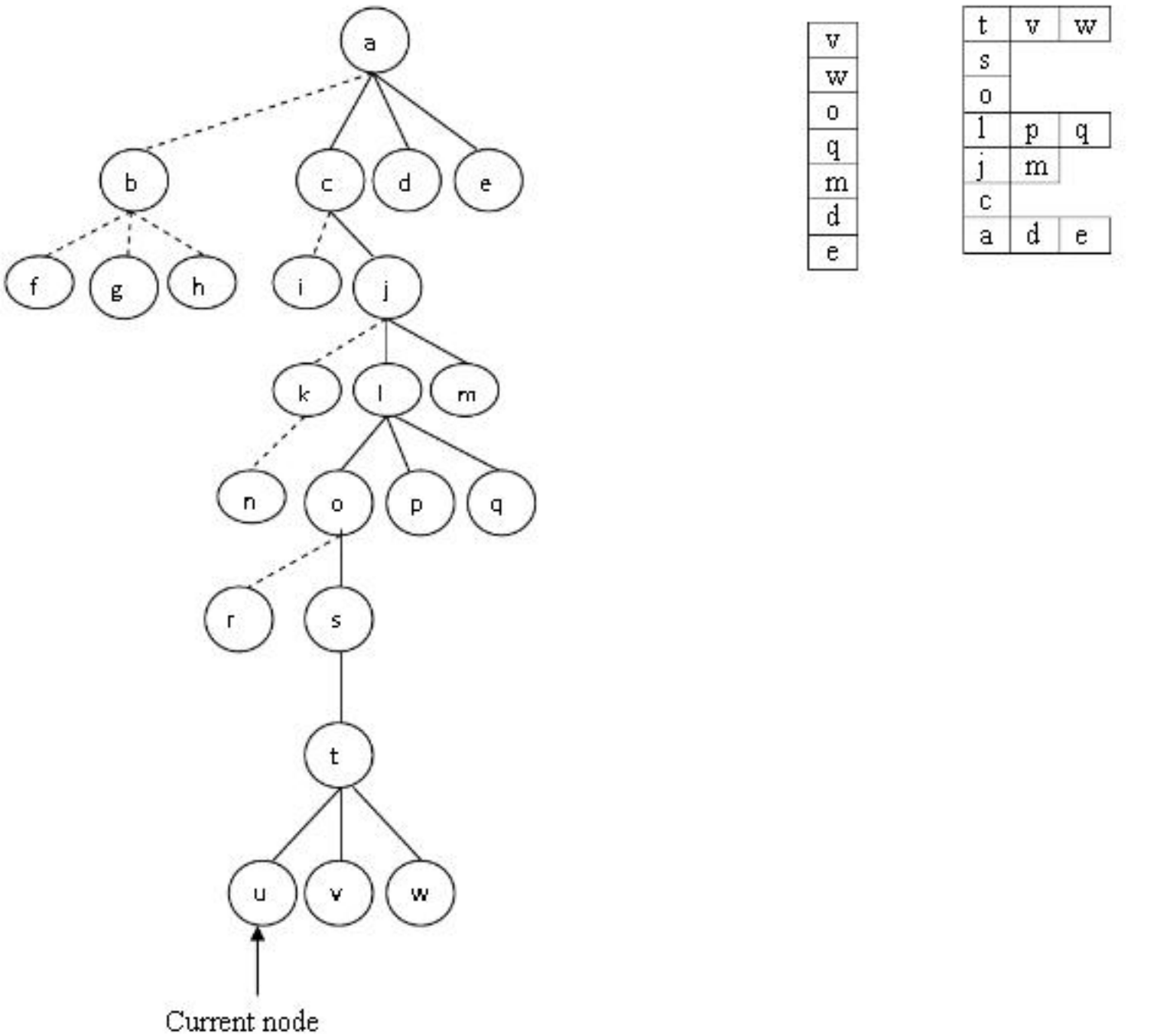
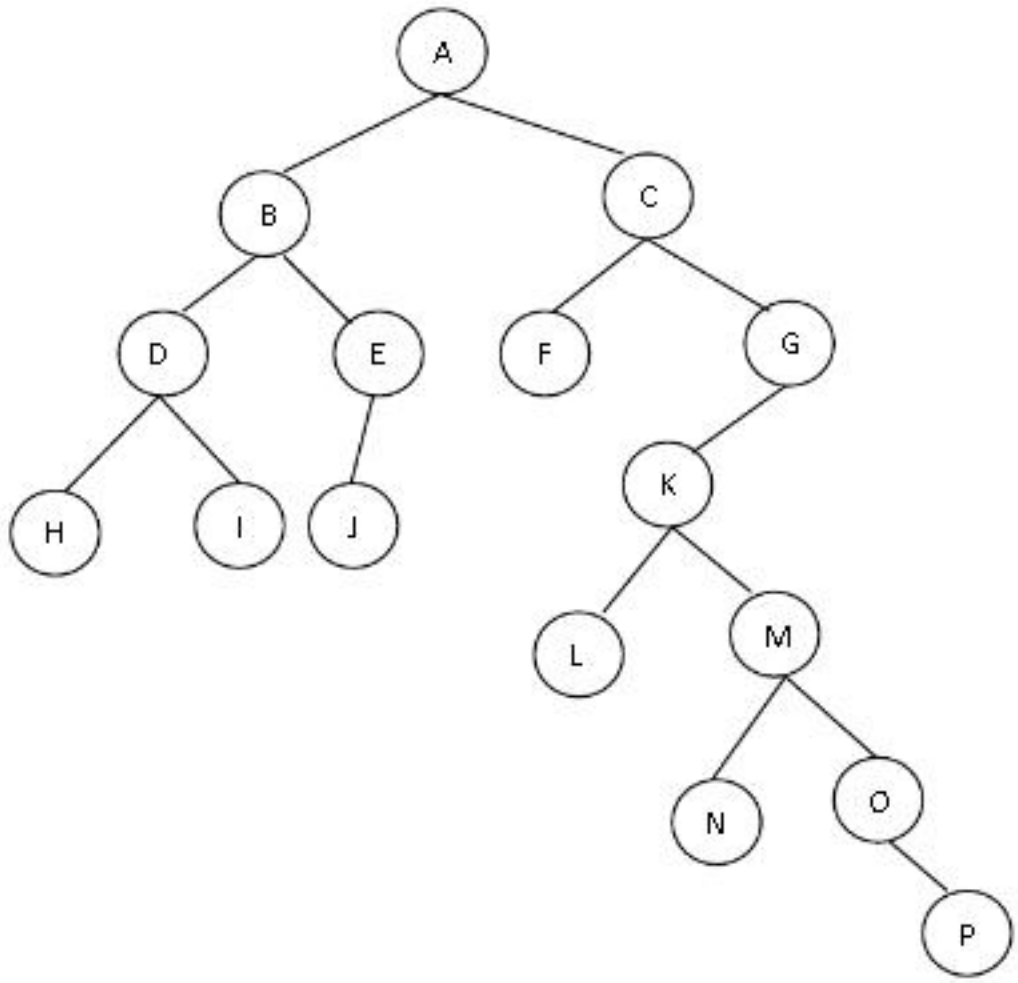
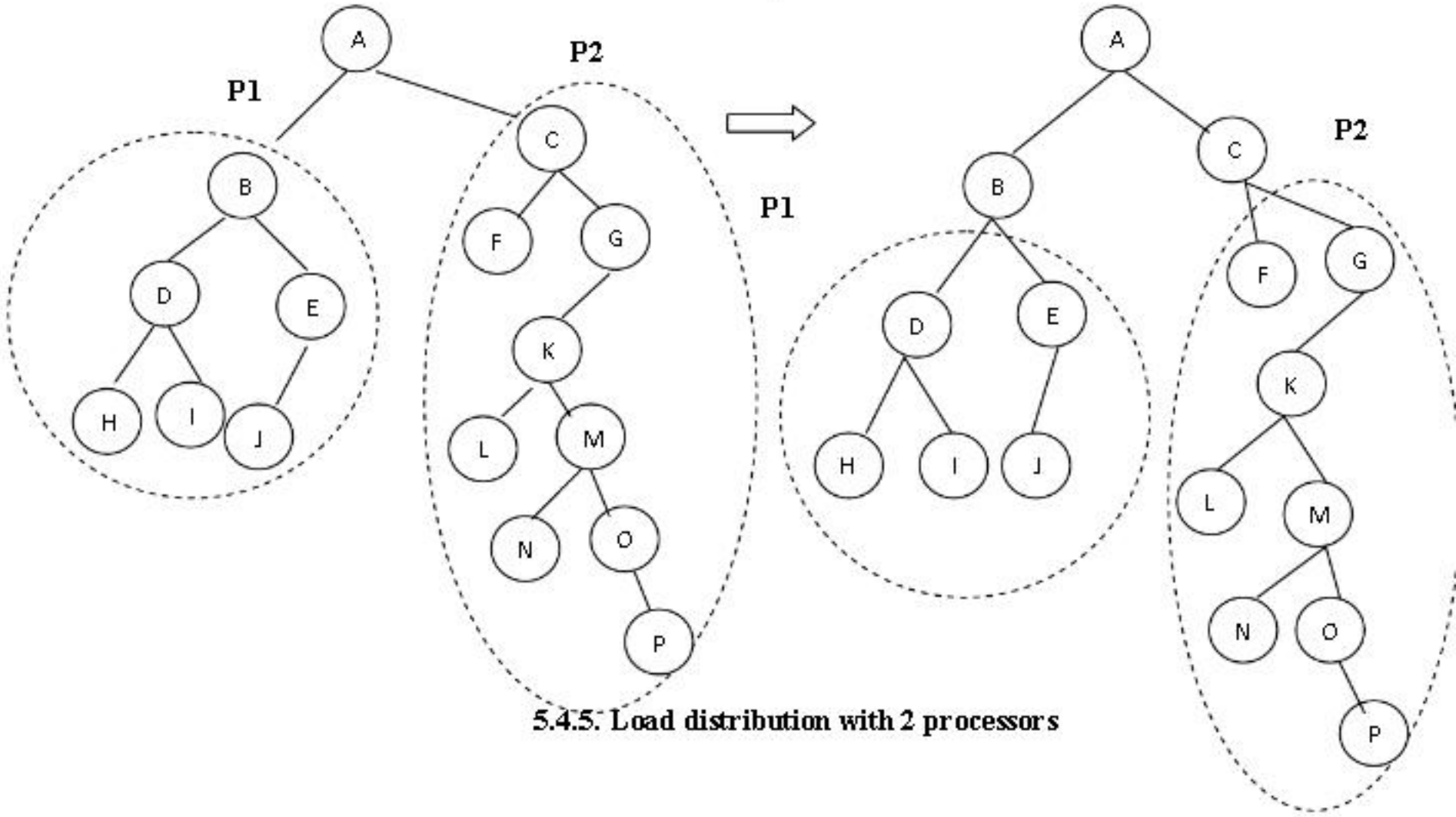


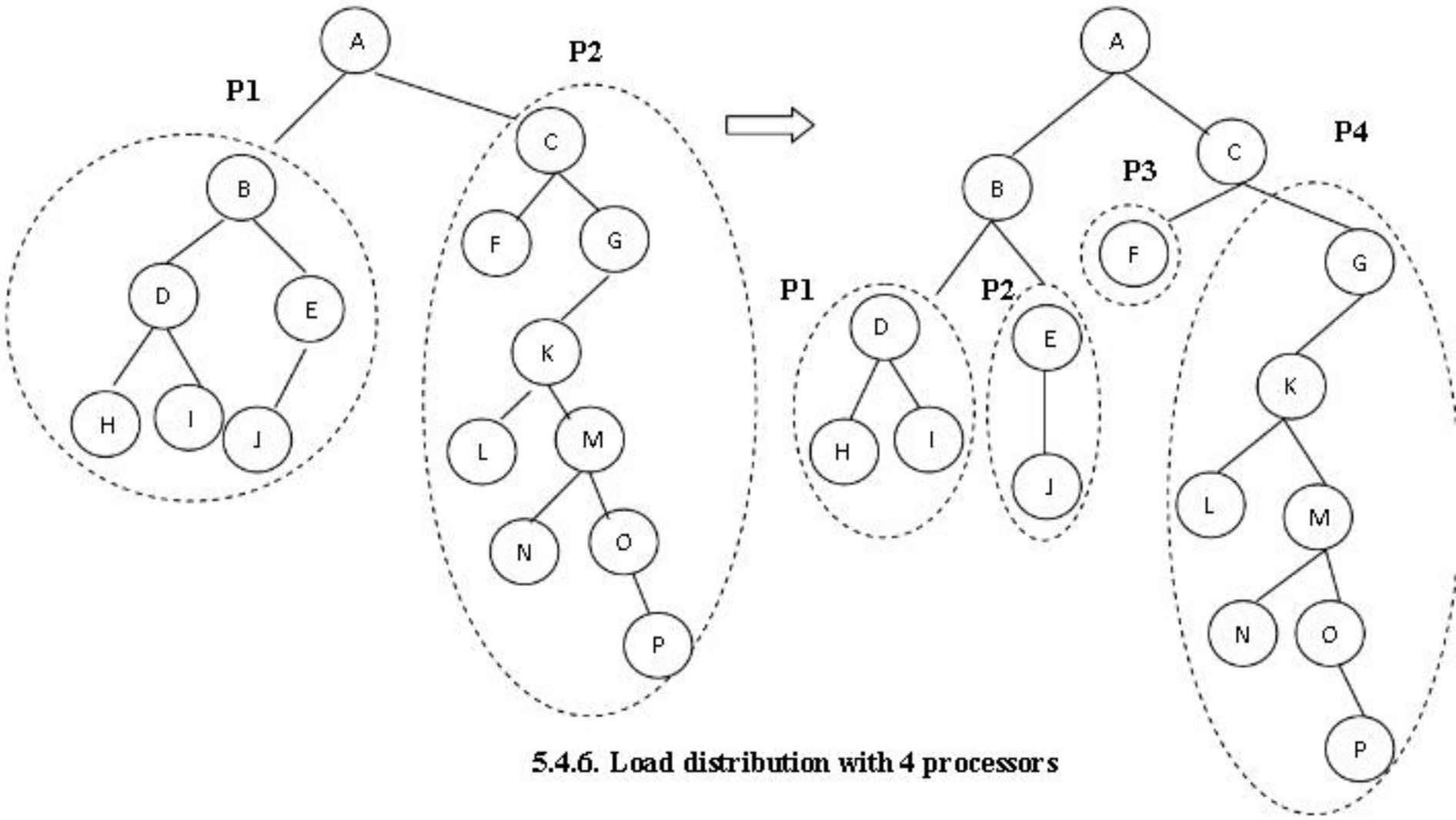
Fig. 5.4.3 Example of DFS tree and corresponding stacks, S1 and S2



5.4.4 Example of DNS tree



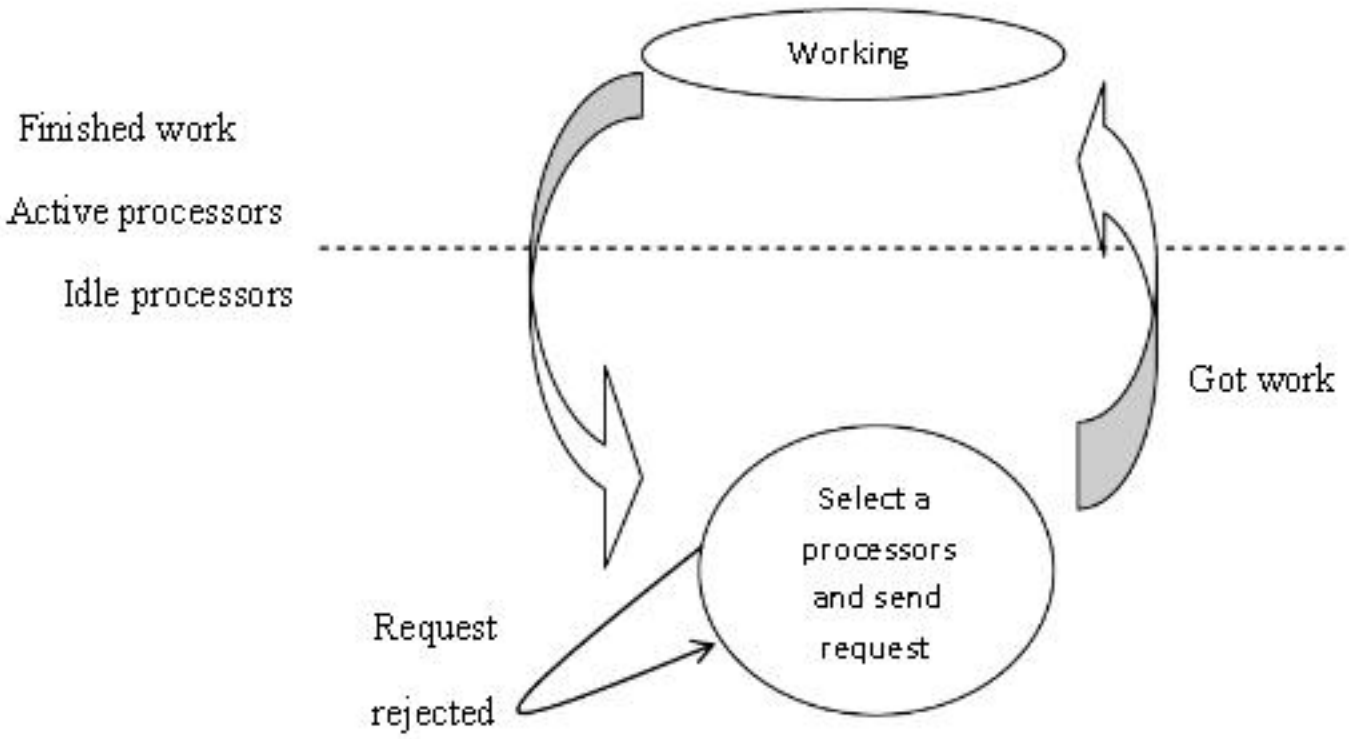
5.4.5. Load distribution with 2 processors



5.4.6. Load distribution with 4 processors

- It is understood that whether there are two processors, or four processors, not all the processors are equally distributed. With two processors, P1 has less load as compared to P2. In case of 4 processors, P4 has the maximum load and P3 has the minimum load.
- Thus, through static distribution seems to be simple it does not make efficient use of processors available. This leads to poor performance. In case of dynamic distribution, an idle processor is allocated work from another processor that has more work.
- A processor, after completion of its work may request for some more work from other processors that are busy.
- This type of distribution reduces the load imbalance among the processors but there is always communication overhead due to work request and work transfer.
- Working of a parallel algorithm, with dynamic distribution for DFS is as follow. Each processor is allocated with a disjoint part of DF search space, in which it performs DFS.
- If any processor finds the foal, it is reported, and all the processors are terminated. If the problem has no solution, then all the processors terminate when the space allocated to them is completely searched.
- When any processor finishes its search without finding a goal then it requests the unsearched parts from other processors.

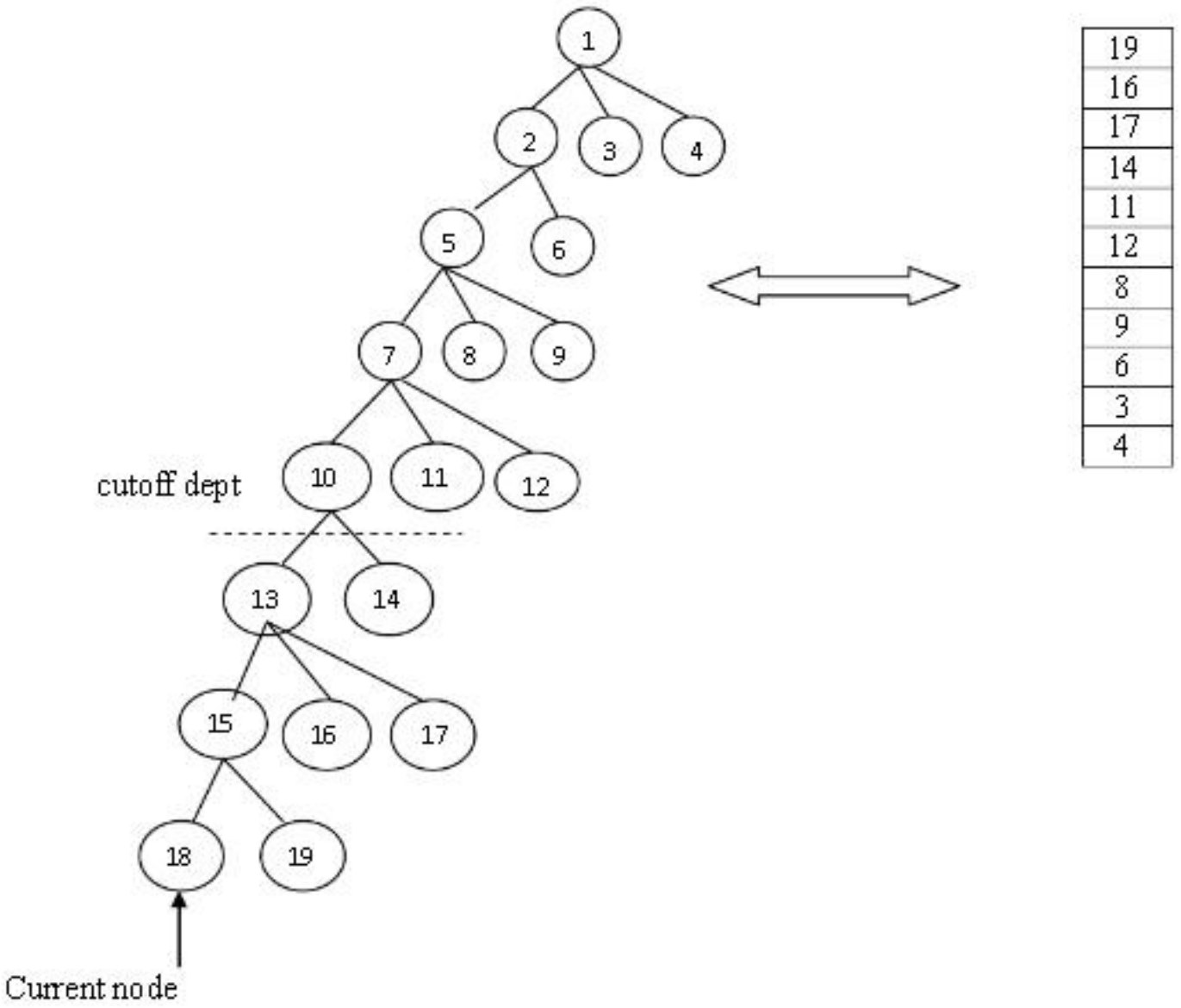
- The complete procedure includes requesting work, response message using message passing interfaces, locking and extracting the work from the shared memory.
- As we have seen in case of sequential DFS algorithm, the unexplored nodes of DFS tree are stored in a stack. Every processor maintains its own stack, which is used to execute DFS.
- Initially the entire search space is allocated to one processor and other processors have empty stacks. The processor with empty stack requests unexplored nodes from the working processor.
- The processor that requests and receives the work is called recipient processor and the processor that sends the work is called donor processor.
- A processor may be in one of the two states, active state which has an allocated work or an idle state which is trying to get work.
- When an idle processor requests for work, it may either get the work and it moves to active state or its request is rejected. When a processor's request is rejected, it requests another processor. It continues until either the request is granted, or all processors become idle. This is shown in Fig. 5.4.7.
- When an active processors gets request from an idle processor, then the work it has must be splitted and given to the requesting processor.



5.4.7 Schematic diagram for dynamic load distribution

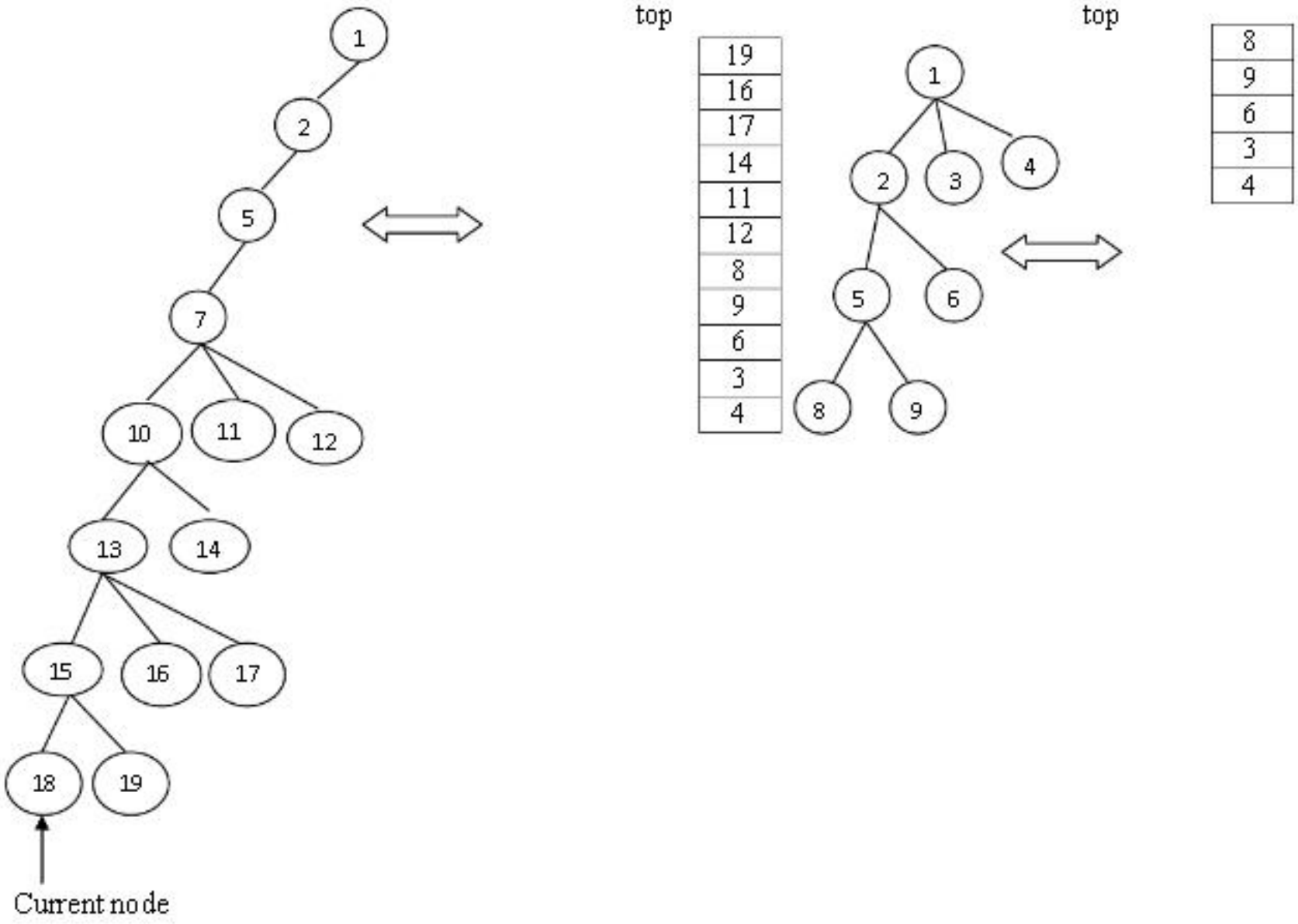
- This is a critical task. There are many strategies used to achieve it. One of the them is called half split. In this case, the stack at the donor is split into two equal parts such that size or the search space represented by each stack is the same.
- Now one of the stack and the corresponding nodes are sent to the recipient. Practically, it becomes difficult to divide the search space into two equal parts. This is because, we cannot estimate the size of the search space without exploring all nodes.

- It is quite obvious that the number of nodes near the root nodes is much lesser than the number of the nodes near the lead nodes.
- Considering this fact, we can have a specific depth in the tree called cutoff depth beyond which the nodes are not given to the recipients.
- Following are the strategies that can be used so that a fair work distribution is done.
  i. Send the nodes near the bottom of the stack
  ii. Send the nodes near the cutoff depth
  iii. Send half nodes between the bottom of the stack and the cutoff depth.
- Selection of a strategy depends on the type of the search space. It is uniformly spread, then we can use (i) or (ii).
- If it is highly irregular, then (iii) would be a better option. If a strong heuristic is available regarding the search space, like goal node is near the left space, then strategy (ii) can be used. When we consider the cost of splitting of stack for a deeper stack strategy (i) has lower cost than the other two.
- Consider the DFS tree in Fig. 5.4.8. The cut-off depth is marked in the tree. The corresponding stack is also shown. The stack contains the unexplored nodes.



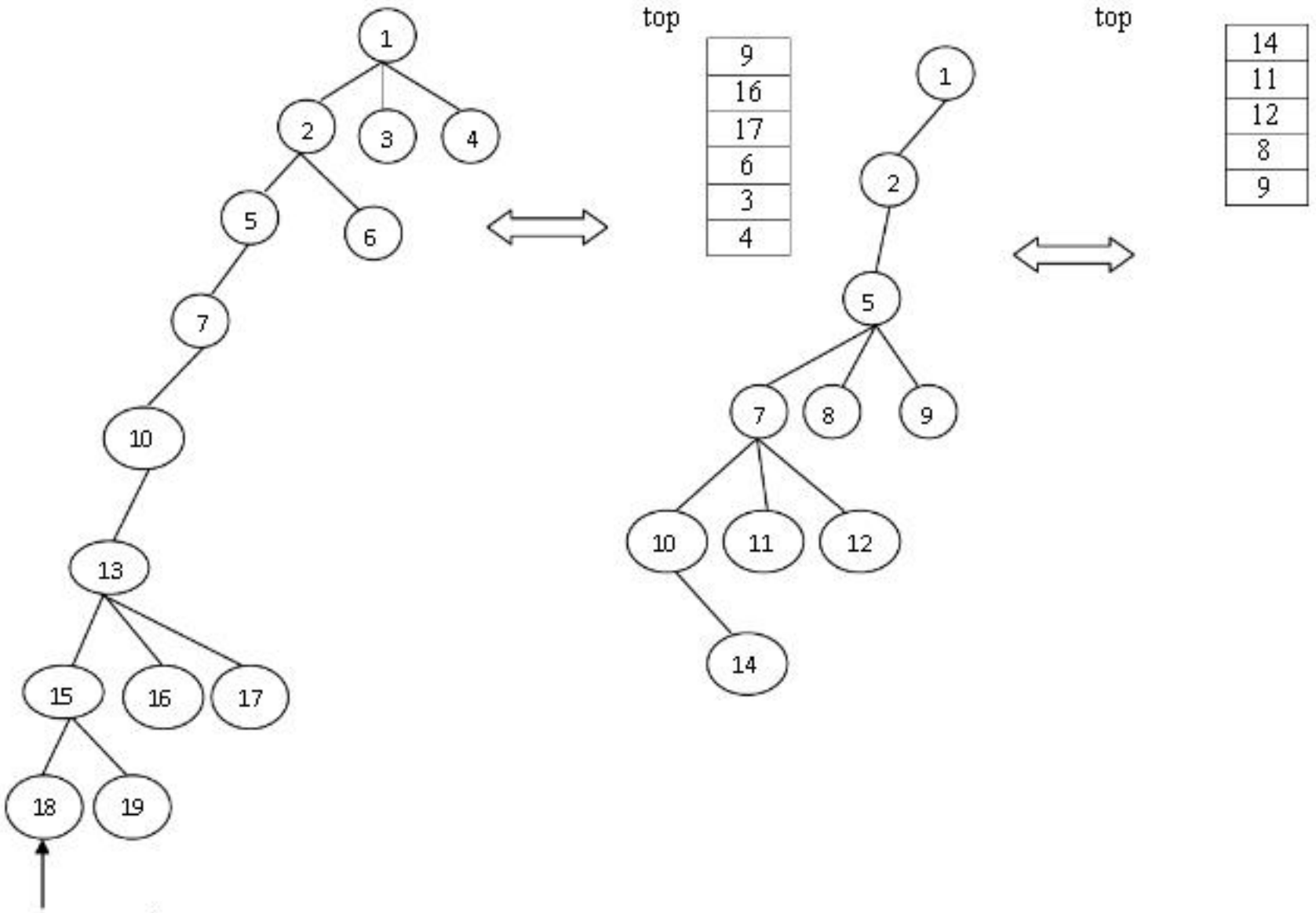
5.4.8 DFS tree and its stack

- If we use strategy (i), then the nodes near the bottom of tree are sent to the recipient. In this case, it would be nodes 8,9,6,3,4. This shown in Fig 5.4.9
- If strategy (ii) is used, then the nodes near the cutoff depth are sent to the recipient.



5.4.9 Splitting of stack near the bottom

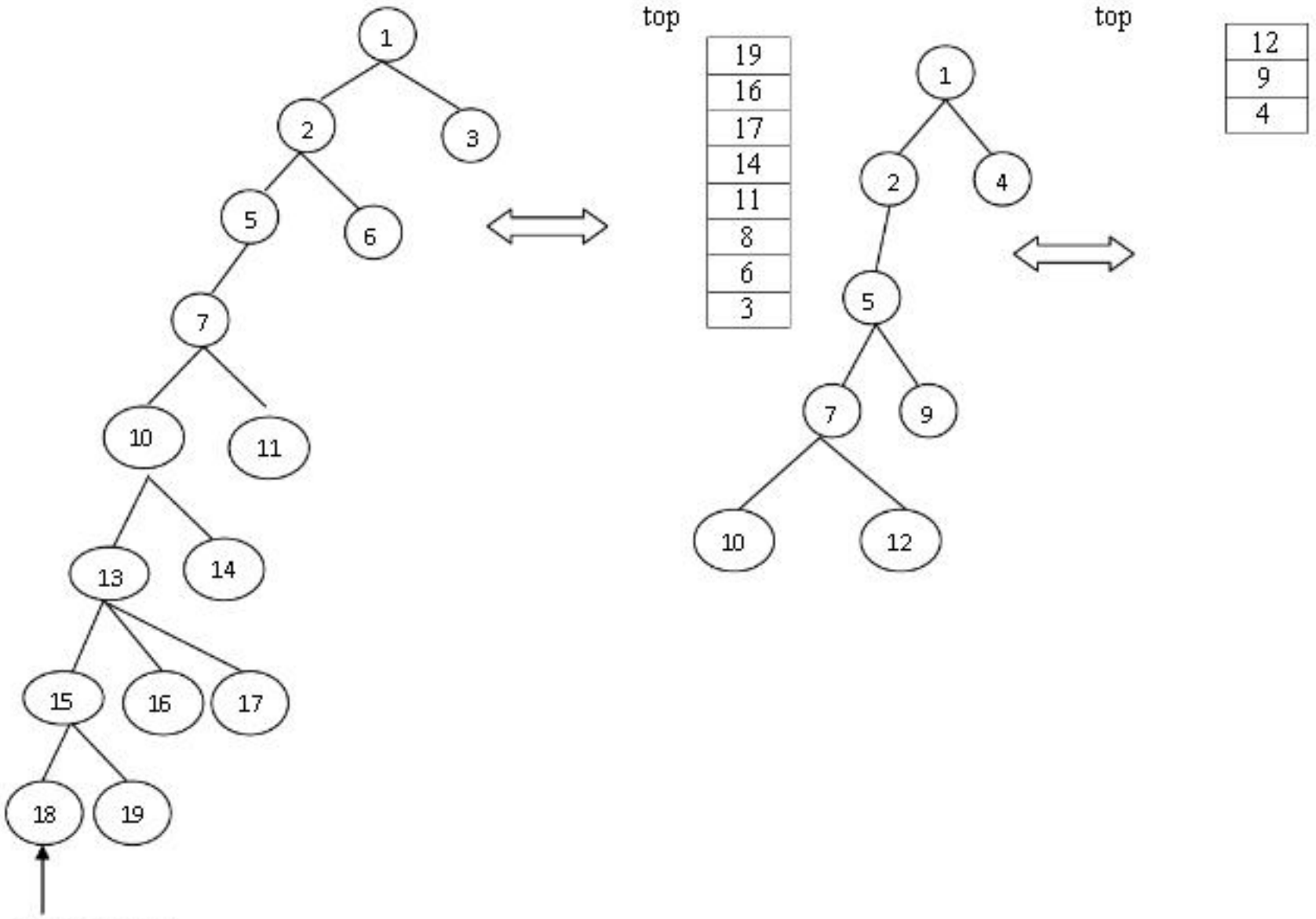
- Thus the nodes near the cutoff depth in this case are 8,9,11,12,14 (we can also consider 11,12,14,16,17). This is shown in Fig. 5.4.10.



Current node

5.4.10 Splitting of stack near the cut off depth

- When the strategy (iii) is used, all the unexplored nodes above the cutoff depth are divided into two equal parts at each level. Then one part is sent to the recipient. In this case there are 4 levels above the cutoff depth. At level one, there are two unexplored nodes 3 and 4.
- So 3 is kept with the donor and 4 is sent to the receipt. This is continues in all other levels. The complete distribution is shown in Fig. 5.4.11.



Current node

5.4.11 Half splitting above cutoff depth

5.4.1 Dynamic Load Balancing Schemes

- When a processor become idle, it has to request for work from any of the active processors. The method by which an active processor is selected is called dynamic load balancing scheme.
- Care should be taken that no processor is repeatedly requested and any processor is unrequested. This is necessary for the uniform work distribution. Following are the different schemes used for load balancing.

i. Asynchronous Round Robin

- In this scheme, each processor has an independent local variable called target. The target of every processor is initialized to zero.
- If there are n+1 processors working parallel, then the processor which is idle may request to one of the n processor.

- The idle processor considers the remaining processors a P0, P1, ..., P_{n-1}. For the first time when a processor becomes idle, it requests from P0, next from P1, and it contains.
- In general, an idle processor requests from processor P_{target}. If the request is rejected, then the request is sent of $P_{(target+1) mod n}$.
- Once the request is granted then the processor increments its target as (target+1) mod n. The work requests are independent of processor and there is a chance that a processor may get request form two or more processors simultaneously.
- In such cases, the donor processor may select any one of them randomly.

ii. Global Round Robin

- In this scheme, a global variable called target is used. When any processor becomes idle, it first requests for the value of target and then it requests for the processor P_{target}.
- The rest of the procedure is same as that of synchronous round robin. After the request is granted, the new value of target is written back.
- If the shared memory address space is used to communicate between processors, then the variable target is stored in the shared memory. When any processor needs it, it must be locked, read and then unlocked.
- If message passing environment is used for communication, then a dedicated processor is used to store the value of target. When any processor needs the target, then a message is sent to this dedicated processor.

iii. Random Polling

- It is the simplest load balancing scheme. When any processor becomes idle, it randomly selects an active processor and sends the request for work.
- If the request is rejected, then another processor is randomly selected and this process continuous until either the request is granted, or all processors are done with their jobs.
- This method has the disadvantage of a chance of a processor getting requested again and again or may not be requested at all. This results into unbalanced load distribution.

5.4.2 Framework for Analysis of Parallel DFS

- The overhead to a parallel algorithm must be computed so as to analyze it, for performance and scalability.
- This overhead is a result of time wasted due to communication, idle time of any processor, contention of shared resources and termination detection of any processor. If search overhead factor is greater than one, then another team must be added to T0. Let us assume that the search overhead factor is equal to one.
- The overhead T0 is given by,

$$T_0 = T_{comm}\ V(P) \log W$$

Where, T_{comm} is the time required for communication, P is the number of processors.

- V(P) is the minimum number of work requests, after which each processor receives at least one work request.

  $$V(P) \geq P$$

  W is the total work to be done.

- Initially processor P_0 as work W, and other processors are idle. Then the idle processors start requesting P_0. If P_1 is granted with the request then the idle processor request either P_0 or P_1, and it continues.
- That is why the term log W is used in the equation, as the work gets reduced after request.
- The efficiency of the parallel DFS is given by,

$$E = \frac{1}{1+T_0/W} = \frac{1}{1+\frac{t_{comm}\, V(P) \log W}{W}}$$

- V(P) for asynchronous round robin lies between p and p^2.
- For global round robin, the value of V(P) is p for random polling, we cannot predict the worst case value of V(P). Thus, it is unbounded but we can have the average case value of V(P) which is equal to O(p log P).

5.4.3 Dijkstra's Token Termination Detection Algorithm

- The simplest case of detecting a termination of a parallel algorithm is a processor goes into idle state and it does not get any work assigned.
- In this algorithm, the n processor P0, P1, ……, P_{n-1} are assumed to be arranged in the form of a logical ring. When any processor Pi becomes idle, it initiates a token and it is sent to its next processor, i.e. $P_{(i+1) \bmod n}$. Any processor that receives a token, holds it until it completes its assigned work. Then the token is passed to its next processor in the logical ring.
- When the processor Pi gets back its token, it is understood that all the processors have completed their assigned tasks. Now the algorithm can be terminated.
- This algorithm, through it is simple, cannot be applied to DFS. As an idle processor have get new work from an active processor. Thus it has to be modified. In the modified algorithm also, the processors are arranged as a ring.
- A processor may be in a white state or black state. Initially all the states are white, When the token is passed through the ring second time, the processor that transmits is marked black.
- The token is marked as black indicating that it is an invalid token, it must be moved through the ring again. When the token becomes white, it indicates it is a valid token. When a white

token is received by the initiating processor, it implies the termination of the algorithm. The detailed working of the modified termination algorithm is as follows.

1. When a processor Pi becomes idle, it initiates termination detection. It marks itself white and a white token is transmitted to processor $P_{(i+1) mod n}$.
2. If a processor Pj sends work to processor Pk, then processor Pj becomes black.
3. If any processor Pl has a token and Pl is black, then it marks the token as black and transmits to the next processor. IF Pl is white, the color of the token is unchanged.
4. When Pl passes the token to $P_{(i+1) mod n}$, Pl becomes white.
5. The algorithm is terminated when the initiating processor Pi is idle, and it receives a white token. It is explained in Fig. 5.4.11.

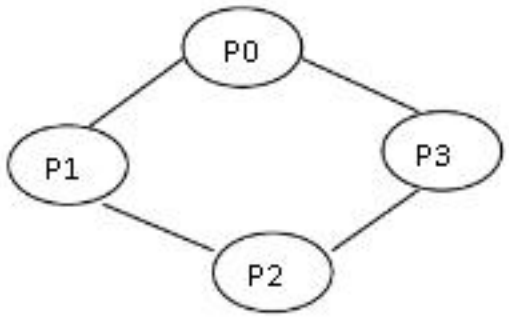
5.4.4 Tree Based Termination Detection

- In this algorithm, every work piece is associated with a weight. As usual, initially the complete work is assigned to processor P0 and the weight associated with it is one.
- When a processor Pi with weight Wi transfers a part of its work to processor Pj, then the weight at pi becomes Wi/2 and weight of Pj is also Wi/2.
- When a processor completes its assigned work, it returns the weight to the donor processor. When the processor P0 finishes its assigned work and it has a weight one, then the algorithm is terminated. It is shown in Fig. 5.4.12.
- The drawback of this algorithm is, as we go on reducing the weight by half, there is a chance that it reaches to zero. Thus termination detection becomes impossible. To avoid this, instead of manipulating Wi we can manipulate 1/Wi.

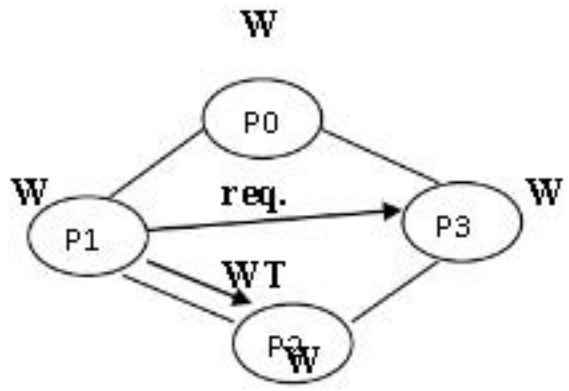
5.5. Parallel Best – First Search

- A Best First Search (BSF) algorithm can be used to search both trees and graph. This algorithm uses heuristics to select the portion of search space.
- The nodes which seem to give the results and assigned with smaller heuristic values than the nodes which are far away from the result.
- The algorithm maintains two lists : an open list and a closed list. Initially, the first node is placed on the open list.
- The algorithm uses a predefined heuristic evaluation function. This function measures how likely a node can yield a solution.
- The open list is sorted as per this function.
- In each step the most promising node from the open list is removed. If it is the required resulting node, then the algorithm is terminated.

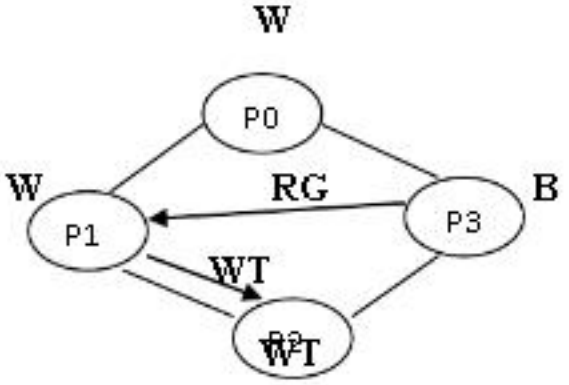
Initially status : all processors are active



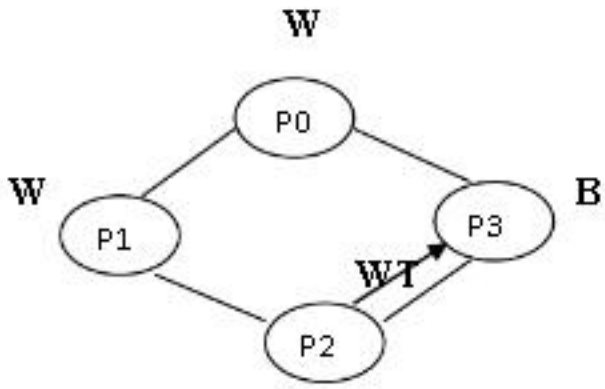
P1 becomes idle. IT sends while token to P2, and request for work to P3 (by using any scheme).



P3 grants request, P3 becomes black. White token is present in P2.

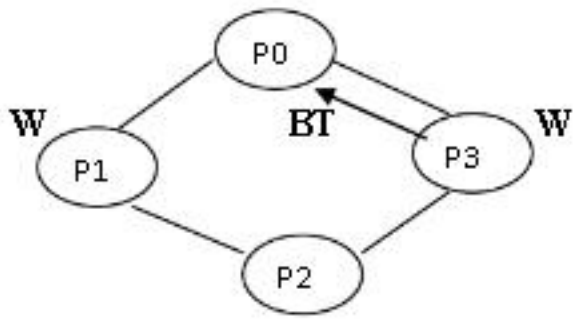


P2 becomes idle, it sends white token to P3



W

P3 becomes idle. It converts token to black and sends to P0, P3 becomes white.



i. Black token from P0 to P1
ii. P1 converts black token to white as it is initiating processor.
iii. (iv),(v) white token comes back to P1.
Algorithm is terminated

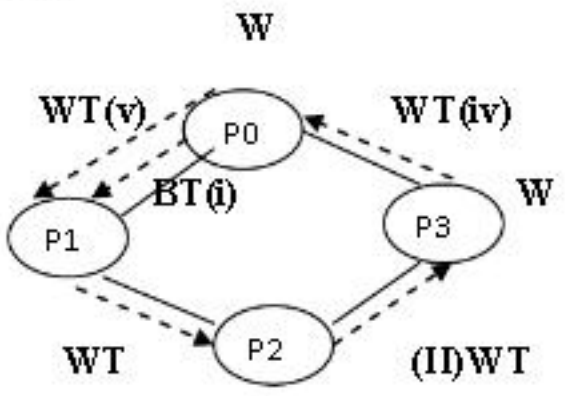
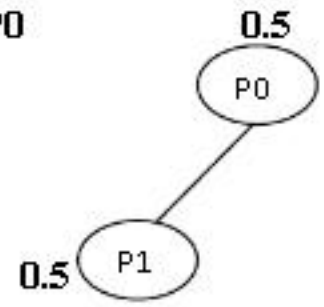


Fig. 5.4.1 Working modified token termination detection algorithm

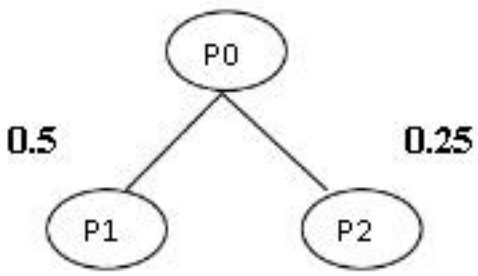
Initial status : one processors with weight 1.

1←weight assigned to processor

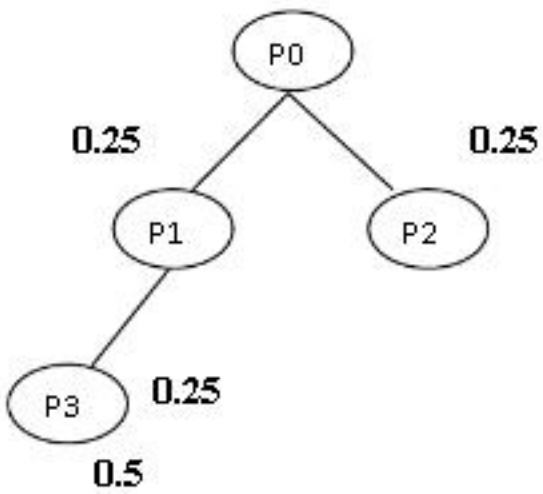
Processor P1 gets work from P0



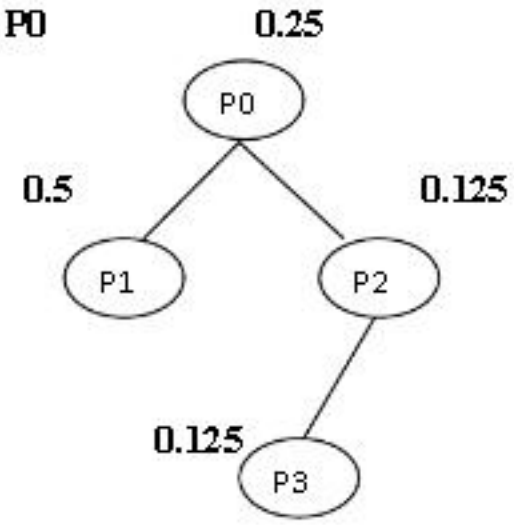
Processor P2 gets work from P0



Processor P3 gets work from P1



Processor P2 gets work from P0



P1 and P3 become idle, get no work

P2 becomes idle, gets no work

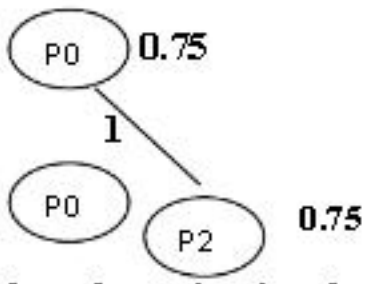
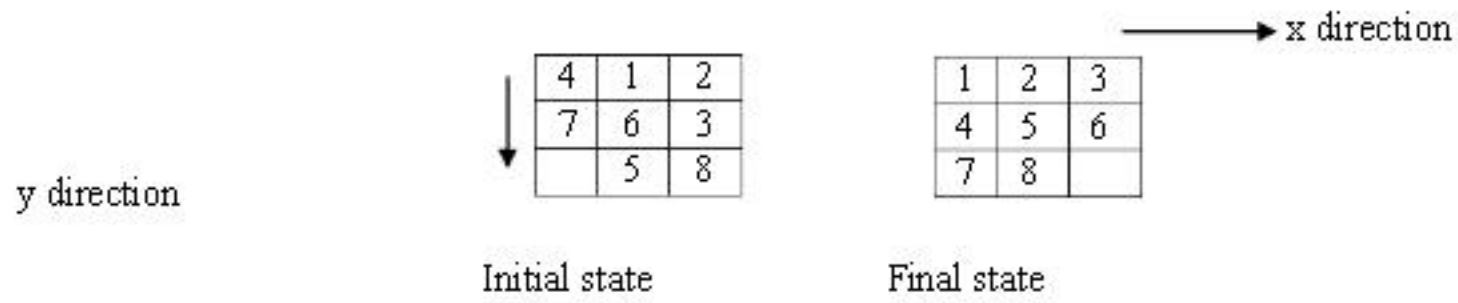


Fig 5.4.2 Working of tree based termination detection algorithm

- When P0 is idle, the algorithm terminates. Otherwise, the node is expanded. Then the expanded node is placed on the closed list. When the node is expanded, it generates its successors.
- If a successor is not present in either of open or closed list, then it is stored in the open list. If a successor is present in open or closed list and its heuristic value within the list is greater than the current node, then the node in the list is detected and new node is inserted.
- If a successor is present in open or closed list and its heuristic value within the list is less than the current node, then no change is done to the list.

- The 8 puzzle can be solved suing BFS algorithm the initial state of the 8-puzzle is given and the final required result is considered. Here the heuristic function to be considered will be the Manhattan distance from a state to the final state.
- When a node is expanded, the successor with the minimum Manhattan distance is selected for further expansion
- Fig. 5.5.3 gives an initial state of 8-puzzle and its final solution state. It also explains the Manhattan distance calculation.
- If more than one successor has minimum Manhattan distance, then any one of them is arbitrary selected.



Initial state Final state

Manhattan distance = $D_1+D_2+D_3+D_4+D_5+D_6+D_7+D_8+D_B$

Where D_i is distance between I in both states.

D_B is distance between blank sequences in both states.

$$D_1 = |2-1| + |1-1| = 1$$

$$D_2 = |2-2| + |1-1| = 1$$

$$D_3 = |2-3| + |2-1| = 1$$

$$D_4 = |2-1| + |1-2| = 1$$

$$D_5 = |2-2| + |3-2| = 1$$

$$D_6 = |2-3| + |2-2| = 1$$

$$D_7 = |2-1| + |2-3| = 1$$

$$D_8 = |2-2| + |3-3| = 1$$

$$D_B = |2-3| + |3-3| = 2$$

Manhattan distance = 10

Fig. 5.5.3 Calculation of Manhattan Distance

- Fig. 5.5.4 gives an example of BFS for 8-puzzle. The initial node has the Manhattan distance 10, and its successors also have the same.
- Thus one of them is selected arbitrarily and expanded. It can be seen that the Manhattan value of a parent node is always greater than or equal to that of the selected successor.
- If at any step, the Manhattan values of all the successors are greater than that of parent, then it must be back tracked. The Fig. 5.5.4 also shows the open and close list at every step of expansion.

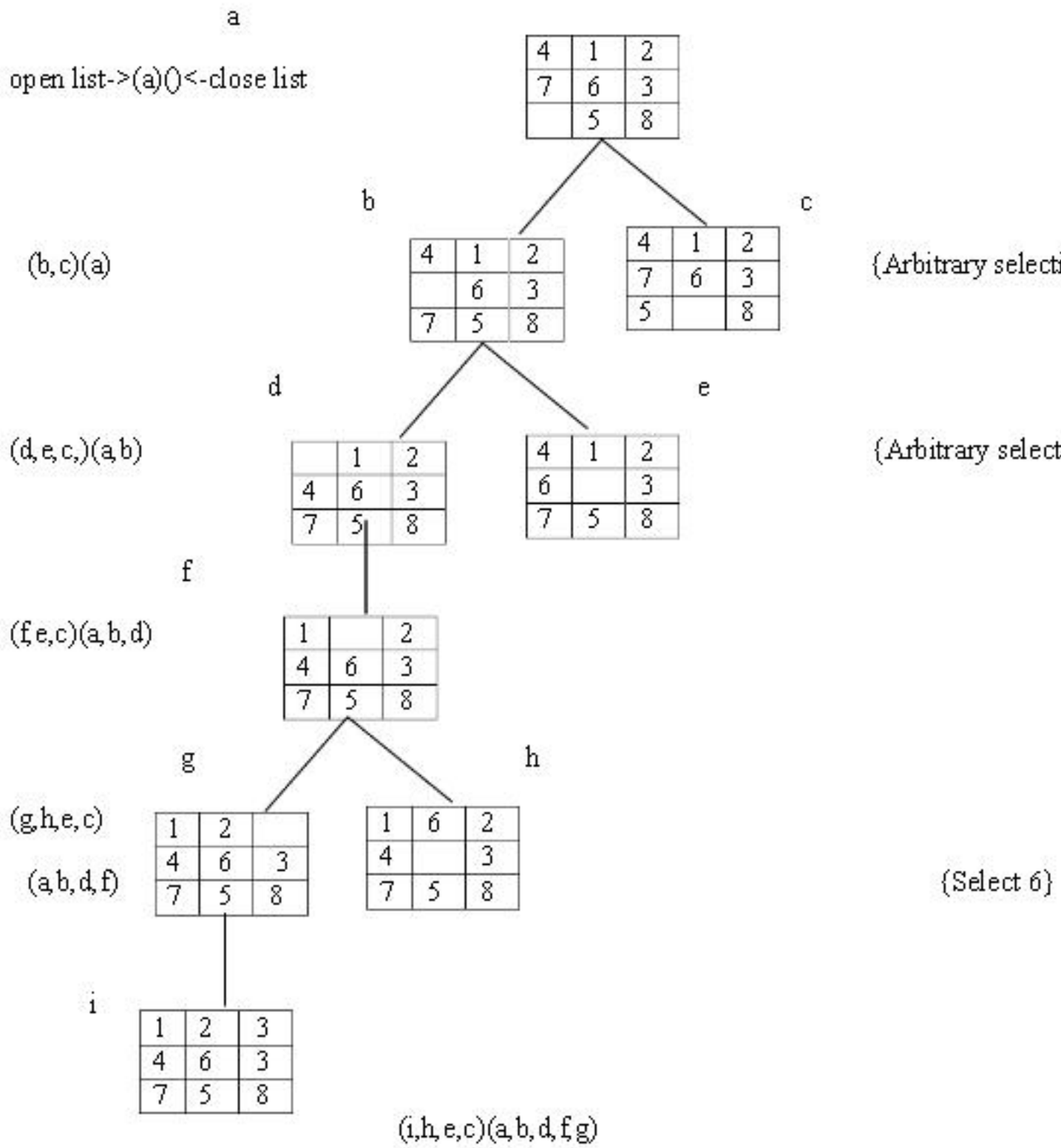


Fig. 5.5.4 BFS example for 8-puzzle along with lists associated at every step

- The open list play an important role in BFS algorithm. In a parallel BFS algorithm, different processors concurrently expand different nodes from the open list.
- The assignment of work to processors is done by centralized strategy. It assigns each processor to work on one of the best current nodes present in a single open list. Hence the name is centralized. Fig 5.5.5 shows the working.
- The disadvantage of this method is finding the condition for termination. As n processors keep on expanding the nodes, if any one processor comes across the goal, it will not be clear until all other processors give their results for comparison.
- Another disadvantage is accessing the global open list. This should address space must be carefully handled so as to do due the work smoothly. The communication strategies are explained in the following subsections.

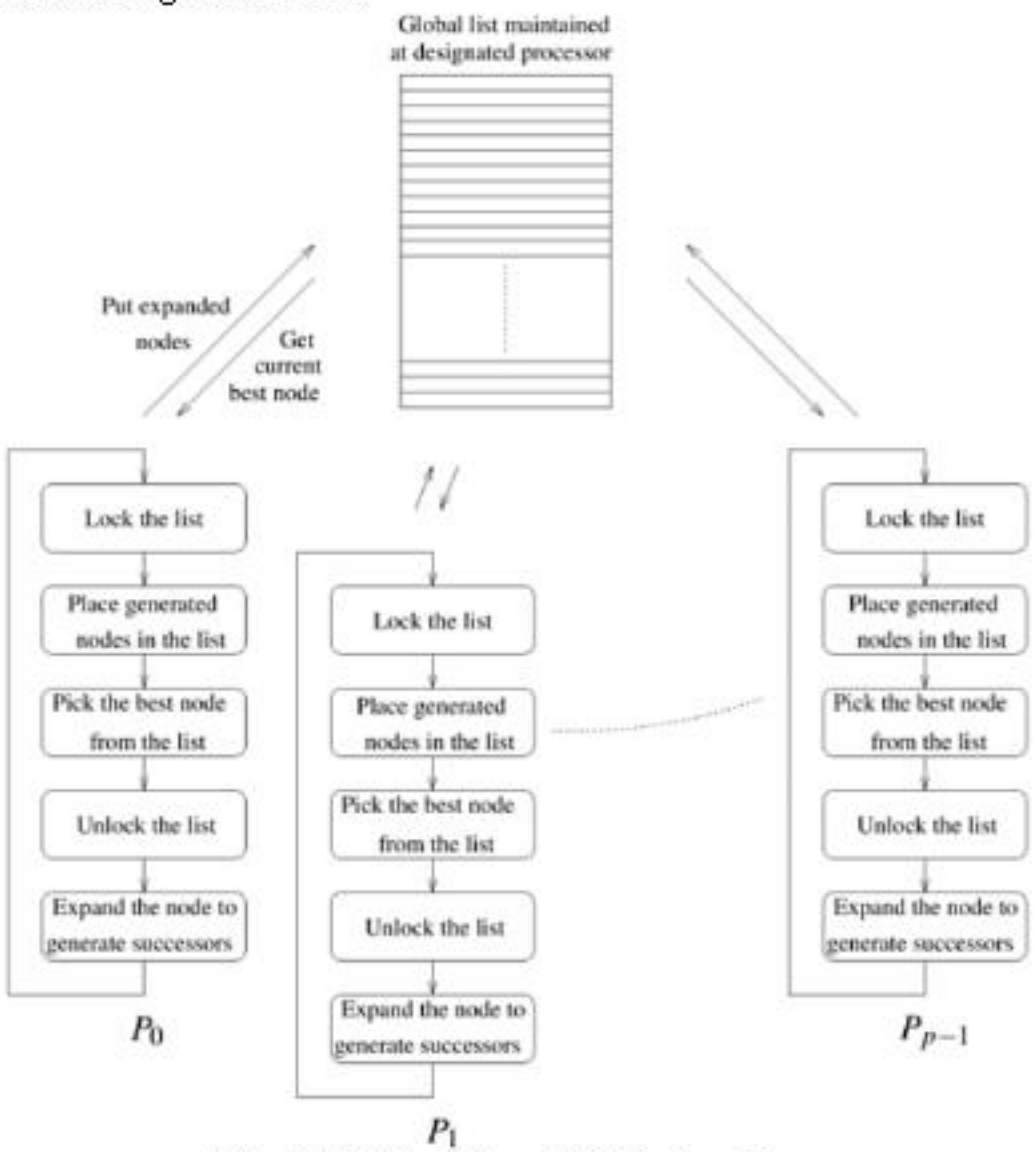


Fig. 5.5.6 Working of BFS algorithm

5.5.1 Random Communication Strategy

- In this type every processor sends some of its best nodes to the open list after a time interval. This open list may belong to any of the randomly selected processor.

- When the nodes are frequently transferred, then the search overhead factor is minimized. If the communication cost is low, then communicating after every node expansion could be the best option. Fig. 5.5.6 explains it.

5.5.2 Ring Communication Strategy

- In this type the processors are arranges as a logical ring. Each processor exchanges its best nodes with its neighbours periodically. This can be implemented using shared memory or message passing.
- If the search space is highly uniform, the search overhead factor is minimum. Fig 5.5.7 explains this.

5.5.3 Blackboard Communication Strategy

- In this type, a shared memory space called blackboard is used. Initially each processor sends its best node to blackboard. Then onwards, every node compares its own best node with the nodes in blackboard.
- If the difference is within a tolerable limit, the node is expanded.
- If the node is much worse than the nodes in blackboard, the some nodes are borrowed from blackboard and a node is reselected.
- If the node is much better than the nodes in blackboard, then it is stored to blackboard and then expanded.

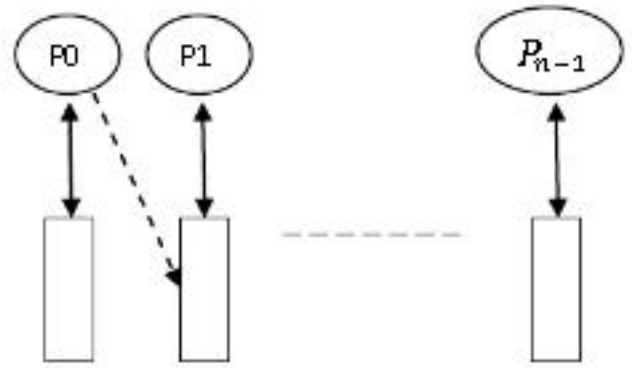


Fig. 5.5.6 Random Communication Strategy

P0, P1, ……., P_{n-1} are processors.

- OL-Open List associated with each processor. Dotted arrow indicates transferring of best nodes of P0, to P1.

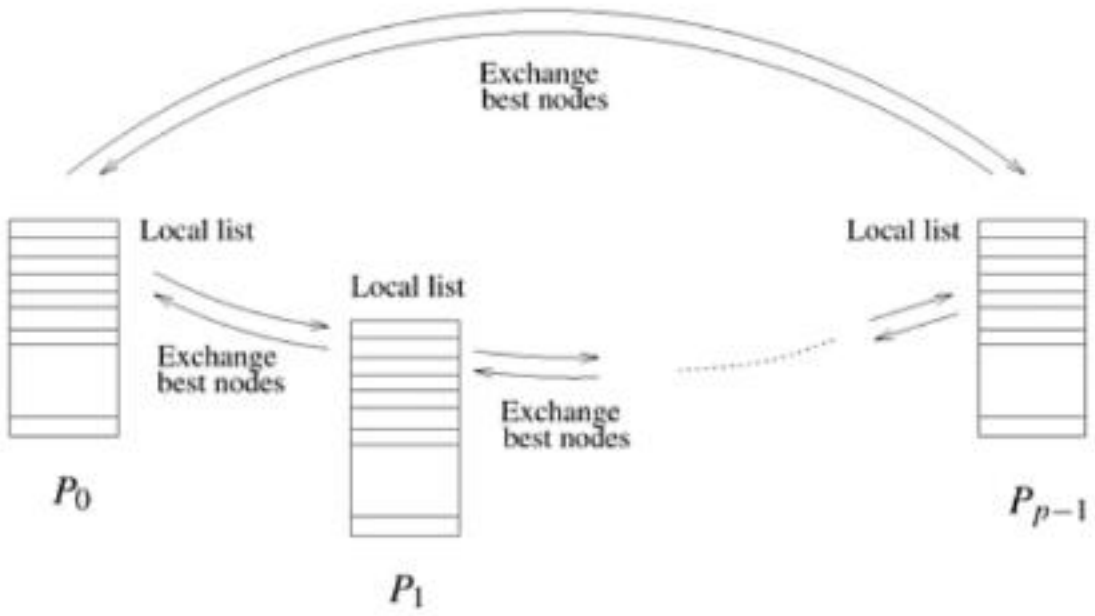


Fig. 5.5.8 Ring Communication Strategy

- Fig. 5.5.7 shows Ring communication strategy by the processor and Fig 5.5.8 explains Blackboard Communication strategy.

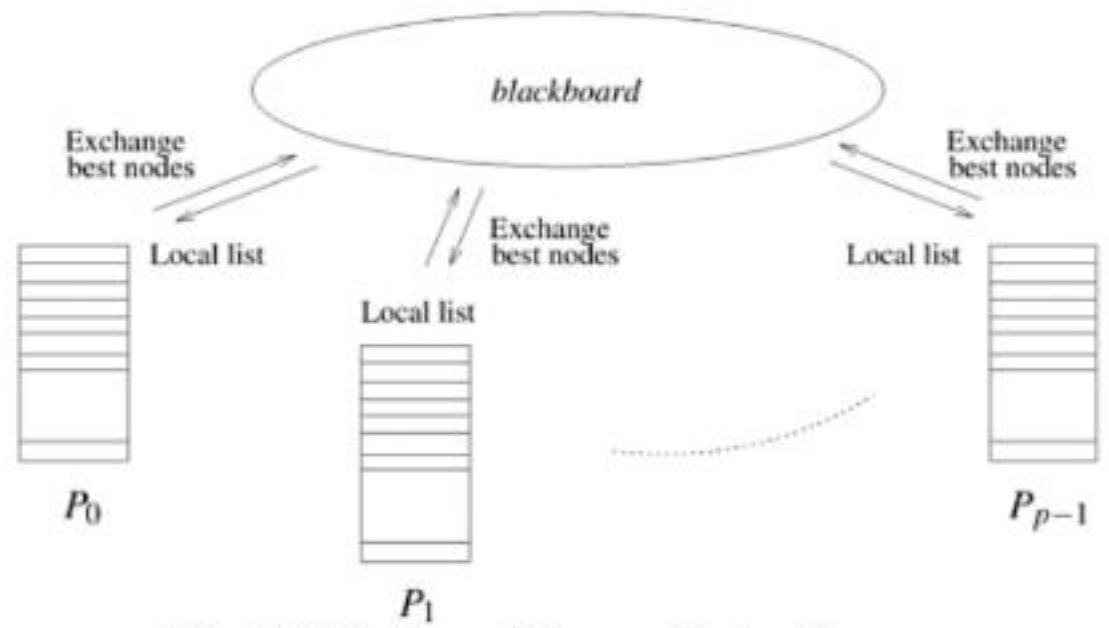


Fig. 5.5.8 Blackboard Communication Strategy

UNIT- VI CUDA Architecture

At the starting of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture. Hence there is a huge need to design and put up the software so that it utilizes multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To put up such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment.

At start, they were utilized for graphics purposes only. But now GPUs are becoming increasingly popular for a variety of general-purpose, non-graphical applications too. For example they are utilized in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc.

6.1 CUDA

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by NVIDIA. It is utilized to put up software for graphics processors and is utilized to put up a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPU's processor cores.

CUDA utilizes a language that is very similar to C language and has a high learning curve. It has some extensions to that language to utilize the GPU-specific features that include new API calls, and some new type qualifiers that apply to functions and variables. CUDA has some specific functions, called *kernels*. A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads. CUDA also provides shared memory and synchronization among threads.

CUDA is supported only on NVIDIA's GPUs based on Tesla architecture.

6.2 The System Model

Graphics processors were mainly utilized only for graphics applications in the past. But now modern GPUs are fully programmable, highly parallel architectures that delivers high throughput and hence can be utilized very efficiently for a variety of general purpose applications.

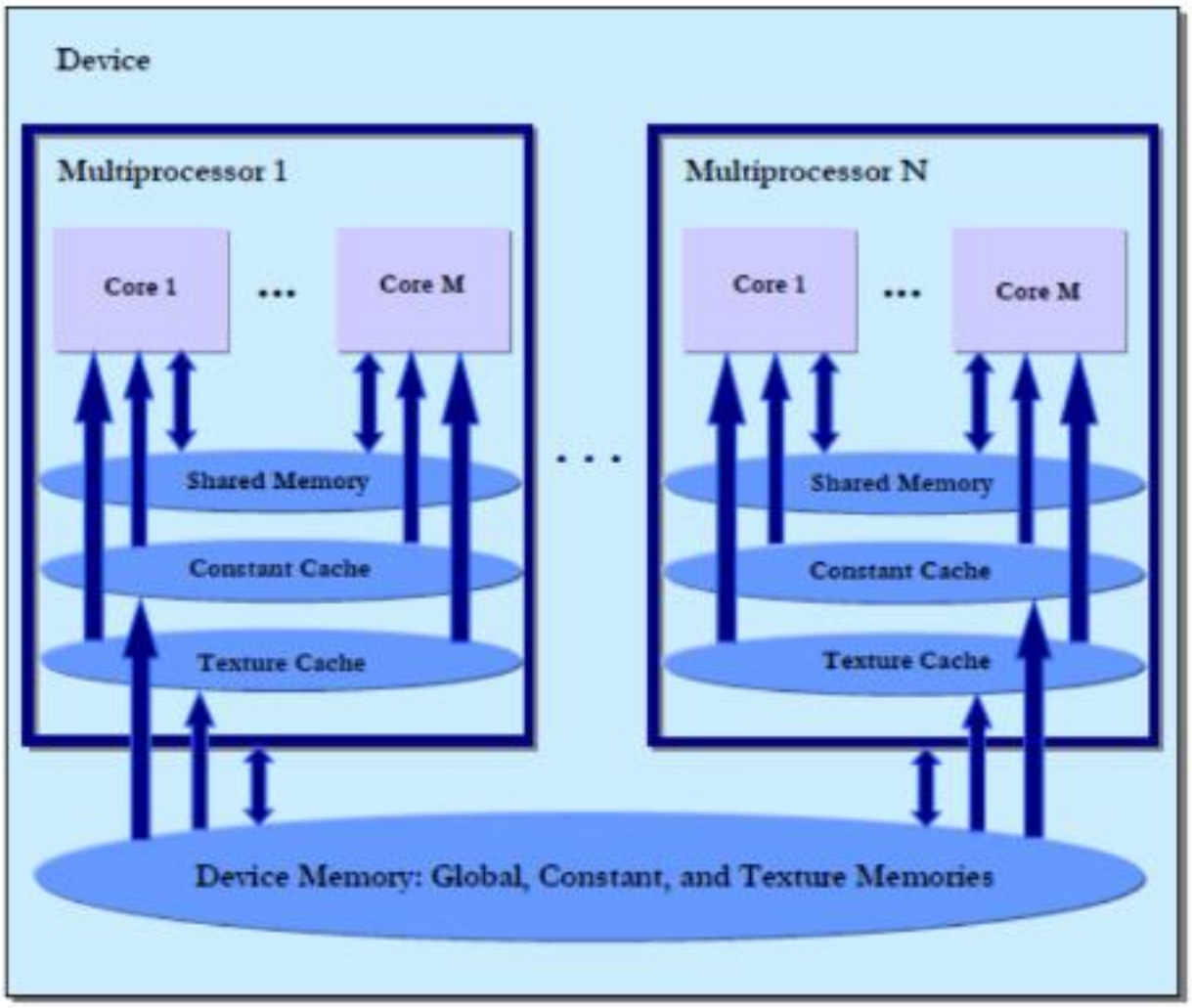


Figure 1: The CUDA Memory Model

NVIDIA's graphics card is a new technology that is extremely multithreaded computing architecture. It consists of a set of parallel multiprocessors, that are further divided into many cores and each core executes instructions from one thread at a time as described in Figure 1.

Hence all those computations in which many threads have to execute the same instruction concurrently, also called data-parallel computations, are well-suited to run on GPU.

NVIDIA has designed a special C-based language CUDA to utilize this massively parallel nature of GPU. CUDA contains a special C function called *kernel*, which is simply a C code that is executed on graphics card on fixed number of threads concurrently. For defining threads, CUDA utilizes a grid structure.

6.3 Heterogeneous Architecture

CUDA programming paradigm is a combination of serial and parallel executions. Figure 2 shows an example of this heterogeneous type of programming.

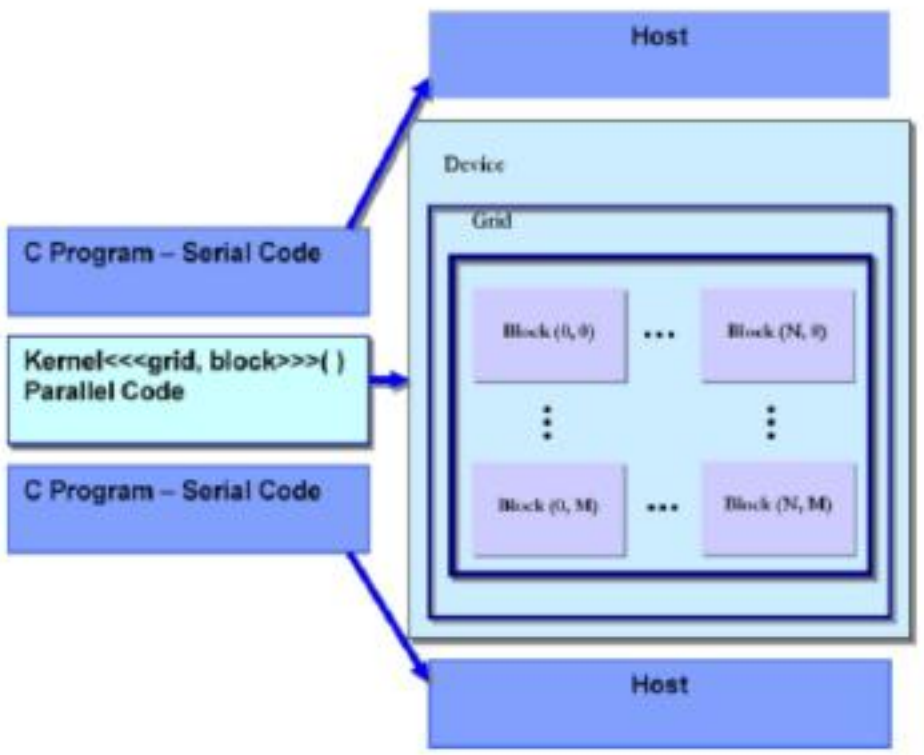


Figure 2: Heterogeneous Architecture

Parallel execution is expressed by the *kernel function* that is executed on a set of threads in parallel on GPU; GPU is also called *device*. This kernel code is a C code for only one thread. The numbers of thread blocks, and the number of threads within those blocks that execute this kernel in parallel are given explicitly when this function is called.

The kernel function can only be invoked by serial code from CPU. To call the kernel function, the execution configuration must be specified, i.e., the number of threads in a thread block and number of threads within a grid. To declare grid and thread blocks CUDA has a predefined data type *dim3*, an integer vector type that specifies the

dimensions of the grid and thread blocks. In the kernel function call grid and block variables are written in three angular brackets <<< grid, block >>> as shown in Figure 2. In this invocation, grid and thread blocks are created dynamically. The value of this grid and block variables must be less than the allowed sizes which are given in next section. The threads are scheduled in hardware and not in software. Kernel function has always a return type *void*. It has a qualifier __global__ that means this is a kernel function to be executed on GPU. See Figure 3 for a graphical description of grid and thread blocks.

6.4 The Grid and block structures

The Grid consists of one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is further divided into one-dimensional or two-dimensional threads. A thread block is a set of threads running on one processor. Figure 3 describes a two-dimensional grid structure and a two-dimensional block structure. Within a thread block, threads are organized together in warps. Normally 32 threads are grouped in one warp. All threads of a warp are scheduled together for execution.

All threads of a single thread block can communicate with each other through shared memory, therefore they are executed on the same multiprocessor. In this way it becomes possible to synchronize these threads.

The CUDA paradigm provides some built-in variables to utilize this structure efficiently. To access the id of a thread block the *blockIdx* variable (values from 0 to gridDim-1) is utilized and to access its dimension the *blockDim* variable is utilized while *gridDim* gives the dimensions of the grid. Each individual thread is identified by *threadIdx* variable, can have values from 0 to blockDim-1. *WarpSize* specifies warp size in the threads. All these variables are built-in in kernel. The maximum allowed sizes of each dimension of grid is 65535, and x, y, and z dimensions of a thread block are 512, 512, and 64, respectively [1] [2].

The allocation of the number of thread blocks to each multiprocessor is dependent on the necessity of the shared memory and registers by each thread block. More memory and registers requirement by each thread block means allocation of less thread blocks to each multiprocessor. In this case the remaining

thread blocks have to wait for their turn for execution.

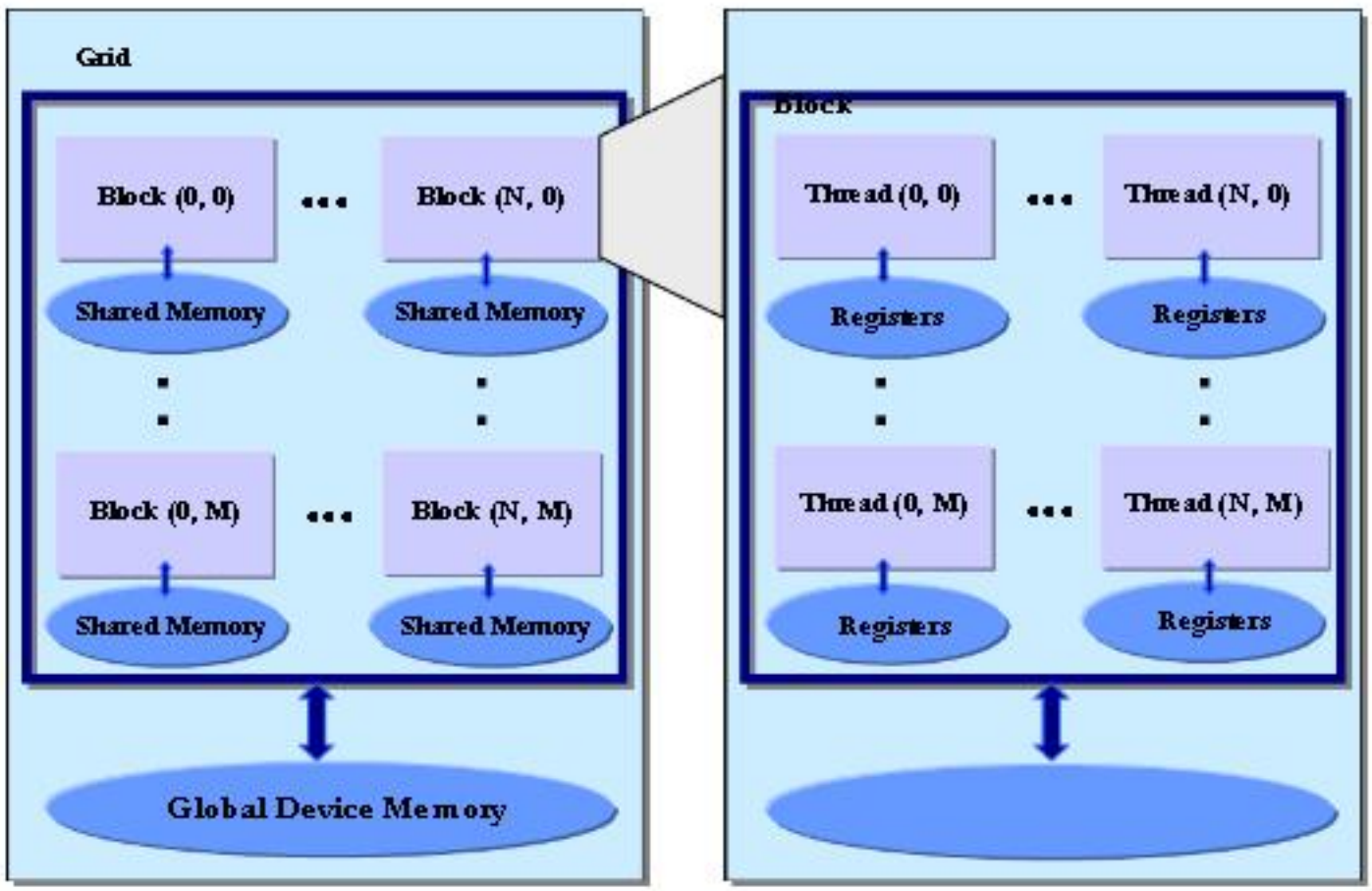


Figure 3: The CUDA Grid Structure and Block Structure.

All this threads creation, their execution, and termination are automatic and handled by the GPU, and is invisible to the programmer. The utilizer only needs to specify the number of threads in a thread block and the number of thread blocks in a grid.

6.5 Memory Model

All multiprocessors access a large global device memory for both gather and scatter operations. Memory model is described graphically in Figure 2. This memory is relatively slow becautilize it does not provide caching.

Shared memory is fast as compared to device memory and normally takes the same amount of time as required to access registers. It is also called parallel data cache (PDC). Shared memory is "local" to each multiprocessor unlike device memory and allows more efficient local synchronization. It is divided into many

parts. Each thread block within multiprocessor accesses its own part of shared memory and this part of shared memory is not accessible by any other thread block of this multiprocessor or of some other multiprocessor. All threads within a thread block that have the same life time as of the block, share this part of memory for both read and write operations. As shared memory space is only 16KiB, so it must be utilized efficiently. To declare variables in shared memory _shared qualifier is utilized and to declare in global memory __device qualifier is utilized.

Each multiprocessor also has its own read only caches to speed up read operation. These are constant cache and texture cache memories.

Each thread also contains its own local memory. Normally local variables of the kernel functions are allocated here. Sometimes they are allocated on global memory.

6.6 Thread Synchronization

For synchronization purpose among threads CUDA API provides a hardware thread-barrier function *syncthreads()* that acts as synchronization point. As threads are scheduled in hardware, this function is implemented in hardware. The threads will wait at the synchronization point until all of the threads have reached at this point. The communication among threads (if required) is possible through per-block shared memory. Hence thread synchronization is possible only at thread block level. Since threads of a thread block may communicate with each other, these threads must execute on same processor. That is why thread block is guaranteed to execute on one processor.

6.7 Number of Threads per Block

To maximize the utilization of available resources, the assignment of the number of threads per block and the number of thread blocks per grid should be done carefully. Less number of threads per block cautilize load latency in device memory reads and also one block per multiprocessor makes the multiprocessor to idle during thread synchronization. Hence there should be at least twice as many blocks as there are multiprocessors in the device (The number of blocks per grid should be at least 100). Also assign the number of threads per block in multiples of the warp size, becautilize it lessens the under-populated wraps.

6.8 Control Flow

As the kernel function runs on the device, memory must be allocated on device in advance before kernel function invocation and if the kernel function has to execute on some data then the data must be copied from the host memory to the device memory. Device memory can be allocated either as *linear memory* or as *CUDA arrays*. Qualifier __device__ at the start of a

variable specifies that space for this variable is allocated on the device memory. CUDA API [2] also has functions to allocate and de-allocate device memory at run time like cudaMalloc(), cudaFree(), etc. Similarly, after the execution of kernel function, data from device memory must be copied back to host memory in order to get results. To copy data to and from the device to host CUDA API provides functions for example cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(), etc. Keeping all this in view the processing flow is as follows:

1. Allocate memory on host and device separately. Device memory is readable and writable by the host through the memory copy functions.
2. Copy data from host to device using CUDA API if required.
3. Kernel function executes parallel on each core.
4. Copy data back from device to host using CUDA API.

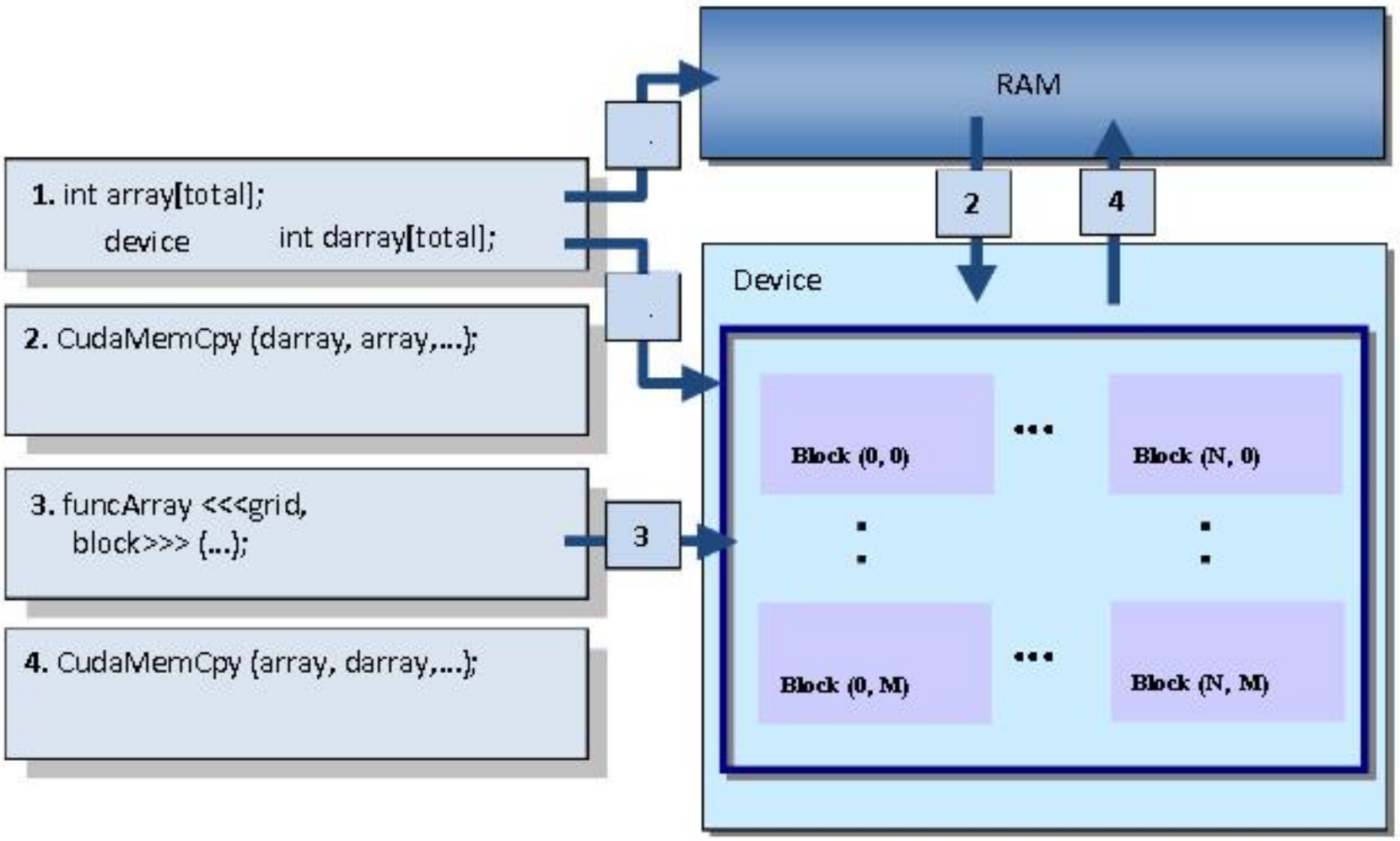


Figure 4: An example of processing flow.

Figure 4 illustrates an example of processing flow of CUDA. In first step two arrays of same size are declared, one on the host and one on the device. The data from the host is copied to the device using CUDA API cudaMemCpy(). The kernel function runs in parallel on the device and in last step the results are copied back to the host using cudaMemCpy() function.

6.9 Transferring Data between Host and Device

Since the bandwidth between the device memory and the host memory is much less as compared to the bandwidth between the device and the device memory which is very high, we should try to minimize data transfer between the host and the device. Some of the efforts could be

like moving some code from the host to the device and creating and destroying data structures in the device memory (instead of copying them to the device) and making huge transfers by batching up many small transfers to lessen the transfer overheads.

6.10 Restrictions

To utilize general purpose GPU we must follow the restrictions of the CUDA programming paradigm. Some of the restrictions are given below:

Simple C programming is supported by the CUDA compiler. It lacks the utilize of object- oriented or C++ features in device code.

Heterogeneous architecture is utilized to make an interaction between CPU and GPU programming models. Data may be copied from host memory to device memory and the results are copied back to host from the device memory. Heterogeneous programming is discussed in section 2.2 and described graphically in Figure 3.

Kernel function invocation: The grid, thread blocks, and threads are created by the kernel function invocation from the host. This is the only way to create them They cannot be created inside the kernel function. The grid, and thread blocks are discussed in Figure 4. Moreover the number of grids and thread blocks must not exceed their maximum allowed values.

The kernel functions do not return any results, i.e. its return type is always void. Further the kernel function call is asynchronous. It means that control returns back before the completion of the kernel function on the device. More information can be found in CUDA programming guide [1]. All functions with the device qualifier are by default inline.

Recursion is simply not allowed within kernel functions becautilize of the large amount of memory requirement for the thousands of thread.

The device memory allocation and de-allocation at run-time is possible only when using host code and before calling the device code. It means that within the device code, the device memory cannot be allocated nor de-allocated using the functions like cudaMalloc(), cudaFree(), etc. All the allocations required for a specific kernel function are done before calling that kernel function in the host code and similarly all that allocated device memory is de- allocated after the completion of that kernel function in the host code.

Shared memory is shared among the threads on the same thread block only. Threads from different thread block cannot share it. This concept is discussed in section 2.4 and graphically shown in Figure 4.

Built-in variables such as blockIdx, threadIdx, etc, cannot be assigned any values. Further it is not possible to take their address.

The variables declared with __device__, __shared__, or __constant__ qualifiers also have some restrictions [1]. Address of a variable with any one of these qualifiers can only be utilized within the device code.

Communication and synchronization among threads are only possible at thread block level. Communication among thread blocks is not allowed. Section 2.5 explains the thread synchronization.

Some Commonly utilized CUDAAPI

1.1 Function Type Qualifiers

The three main types of the funcon qualiers in CUDA are *device, global, and host.*

1. **__device__**

   The functions with device qualifier are executed on the device. These functions are callable from the device only.

2. **__global__**

   The functions with global qualifier are executed on the device but they are callable from the host only.

3. **__host__**

   The functions with host qualifier are executed on the host and are callable from the host only. When no qualifier is utilized, it means that the function will run on the host; it is equivalent to the function declared with the __host__ qualifier.

1.2 Variable Type Qualifiers

The three main types of the variable qualiers in CUDA are *device, constant, and shared.*

1. **device**
   The variables declared with device reside on the device. Other type qualifiers are optionally utilized together with device . If a variable is declared only with device qualifier then this variable resides in the global memory and it has the lifetime of the application. Since it resides in the global memory, it is accessible from all the threads (within the grid) and host through the runtime library.
2. **constant**

   This qualifier is utilized to allocate constants on the device. It is optionally utilized together with device qualifier. This constant resides in constant memory, and has the lifetime of an application. It is accessible from all the threads (within grid) and host through the runtime library.
3. **shared**

   This qualifier is utilized to allocate the shared variable. It is optionally utilized together with
   device qualifier. Shared variable resides in shared memory of a thread block, and has the lifetime of a block. It is only accessible from all the threads within the block.

1.3 Built-in Variables

Following is a list of some of the built-in variables in CUDA:

1. **gridDim:** is of type **dim3** and consist of the dimensions of the grid.
2. **blockIdx:** is of type **uint3** and consist of the block index within the grid.
3. **blockDim:** is of type **dim3** and consist of the dimensions of the block.
4. **threadIdx:** is of type **uint3** and consist of the thread index within the block.
5. **warpSize:** is of type **int** and consist of the warp size in threads.

1.4 Memory Management

1. Memory Allocation

```
float* darray;
cudaMalloc((void**)&darray, 1024 * sizeof(float));
```

2. Memory Deallocation

```
cudaFree(darray);
```

1.5 Copying Host to device

1. Copying host memory array to device memory:

```
cudaMemcpyToSymbol( const T& symbol, const
void* src, size_t count)
```

```
Example:
float cpuArray [1024];
_device_ float dArray [1024];
cudaMemcpyToSymbol (dArray, cpuArray, sizeof(cpuArray));
```

2. Another method

Example:

```
float cpuArray[1024];
int size = sizeof(cpuArray);
float* dArray;
cudaMalloc((void**)&dArray,
size);
cudaMemcpy(dArray, cpuArray, size,
cudaMemcpyHostToDevice);
```

2. Copying host memory array to constant memory:

Example:

```
__constant float
constArray[1024]; float
cpuArray[1024];
cudaMemcpyToSymbol(constArray, &cpuArray,
sizeof(constArray));
```

1.6 Copying Device to Host

1. Copying device memory array to host memory:

```
cudaMemcpyFromSymbol( void *dst, const T&
symbol, size_t count)
```

```
Example:
float cpuArray [1024];
_device_ float dArray [1024];
cudaMemcpyFromSymbol (&cpuArray, dArray, sizeof(dArray));
```

2. Another method

Example:

```
float cpuArray[1024];
int size =
sizeof(cpuArray);
float* dArray;
cudaMalloc((void**)&dArray, size);
cudaMemcpy(cpuArray, dArray, size,
cudaMemcpyDeviceToHost);
```

1.7 Device Runtime Component

Device runtime components are only be utilized in the device functions and are prefixed with an underscore symbol_. The following is a short list of these functions:

1. Mathematical Functions:

   (e.g._sinf(x) ,_cosf(x), sqrt(x), etc)
2. Synchronization Function:

   void_syncthreads();
3. Atomic Functions:

   (e.g. atomicAdd(), etc.)
4. Texture Functions:

1.8 Device Emulation Mode

A device emulation mode is provided basically for the debugging purpose. –deviceemu option is utilized with nvcc compile command. It only emulates the device, it is not the simulation. Threads and the thread blocks are created on the host. Host's native debugging (like Microsoft Visual studio's) can be utilized in setting break points and data inspection. It is especially helpful in input or output operations to the files or to the screen, like the utilize of printf() function, that is not possible to run on the device.

1.9 An Example

3.9.1 Sequential Code

A sequential program to calculate the distances from a specific point to the all other points in a 2D Matrix of order N × N is given below:

```
const int
N=16; void
main (void) {

  int i, j, x, y;

  float hgrid[N][N];
```

```
printf( "\n\tEnter the x coordinate of node : " );
scanf_s("%d", &x); printf( "\n\tEnter the y
coordinate of node : " ); scanf_s("%d", &y);


// Code to find distance without
using device for (i=0; i<N; i++){

        for (j=0; j<N; j++) {

                n = ((i-x)*(i-x))+((j-y)*(j-y)); // distance
                formula hgrid[i][j] = sqrt(n);   //
                distance     formula     printf("\t%.0lf",
                hgrid[i][j]);

        }

        printf("\n\n");

}
```

3.9.2 Parallel Code – 1D Grid

Now the same program is converted to the parallel code to run on the device. A one dimensional grid with only one thread block is utilized. The thread block contains 16 * 16 threads (hence 256 threads in total) in a two dimensional form

```
const int N=16;

__device float dgrid[N][N]; // array on device memory


// function on device to calculate distance

__global void findDistance( int
  x, int y){ int i = threadIdx.x;

  int j = threadIdx.y;


  float n = ((i-x)*(i-x))+((j-
  y)*(j-y)); dgrid[i][j] =
  sqrt(n);

}



void main
  () { int i,
  j;

  float hgrid[N][N];


  dim3 dBlock(N, N);        // thread block with total 256 threads
```

```
printf( "\n\tEnter the x coordinate of node : " );
scanf_s("%d", &i); printf( "\n\tEnter the y
coordinate of node : " ); scanf_s("%d", &j); printf(
"\n\tDistance from a node!\n\n\n" );


findDistance<<<1, dBlock>>>(i, j); // Calling kernel function

cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy device
memory to host


printf( "\n\n\tValues in
hgrid!\n\n" ); for (i=0; i<N;
i++){

  for (j=0; j<N; j++)

        printf("\t%.0lf",
        hgrid[i][j]);
        printf("\n\n");
      }

    }

}
```

3.9.3 Parallel Code – 2D Grid (2 * 2)

Now the same program is converted to the parallel code to run on the device with a two dimensional grid (2 thread blocks in x dimension and 2 in y dimension). The thread block contains 16 * 16 threads (hence 256 threads in total) in a two dimensional form. Hence total 1024 threads will run in parallel in the device.

```
const int
N=16; const
int D=2;

__device float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
 __global void findDistance( int x, int y){
   int i = blockIdx.x * blockDim.x +
   threadIdx.x; int j = blockIdx.y *
   blockDim.y + threadIdx.y;

   float n = ((i-x)*(i-x))+((j-
   y)*(j-y)); dgrid[i][j] =
   sqrt(n);
}

void main
   () { int i,
   j;
  float hgrid[N*D][N*D];
```

```
dim3 dGrid(D,D);            // 2D grid with total 4 thread
blocks dim3 dBlock(N, N);       // thread block with
total 256 threads


printf( "\n\tEnter the x coordinate of node : " );
scanf_s("%d", &i); printf( "\n\tEnter the y
coordinate of node : " ); scanf_s("%d", &j); printf(
"\n\tDistance from a node!\n\n\n" );


findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy
device memory to host


printf( "\n\n\tValues in
hgrid!\n\n" ); for (i=0; i<N*D;
i++){

  for (j=0; j<N*D; j++)
            printf("\t%.0lf",
            hgrid[i][j]);
            printf("\n\n");
      }
   }


}
```

*3.9.4 Parallel Code – 2D Grid (4 * 4)*

The same program is converted to the parallel code to run on the device with a two dimensional grid (4 thread blocks in x dimension and 4 in y dimension). The thread block contains 8 * 8 threads (hence 64 threads in total) in a two dimensional form. Hence total 1024 threads will run in parallel in the device.

```
const int
N=8;
const int
D=4;

__device float dgrid[N*D][N*D]; // array on device memory

// function on device to calculate distance
 __global void findDistance( int x, int y){
   int i = blockIdx.x * blockDim.x +
   threadIdx.x; int j = blockIdx.y *
   blockDim.y + threadIdx.y;

   float n = ((i-x)*(i-
   x))+((j-y)*(j-y));
   dgrid[i][j] = sqrt(n);
}

void
   main
   () {int
   i, j;
```

```
float hgrid[N*D][N*D];

dim3 dGrid(D,D);            // 2D grid with total 16
thread blocks dim3 dBlock(N, N);    // thread block
with total 64 threads

printf( "\n\tEnter the x coordinate of node : " );
scanf_s("%d", &i); printf( "\n\tEnter the y
coordinate of node : " ); scanf_s("%d", &j);
printf( "\n\tDistance from a node!\n\n\n" );

findDistance<<< dGrid, dBlock>>>(i, j); // Calling kernel function
cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid)); //copy
device memory to host

printf( "\n\n\tValues in
hgrid!\n\n" ); for (i=o;
i<N*D; i++){

  for (j=o; j<N*D; j++)
        printf("\t%.olf",
        hgrid[i][j]);
        printf("\n\n");

  }
}

}
```