



BIJU PATNAIK UNIVERSITY OF TECHNOLOGY,  
ODISHA

Lecture Notes

On

**PARALLEL COMPUTING**

Prepared by,  
Dr. Subhendu Kumar Rath,  
BPUT, Odisha.

# PARALLEL COMPUTING

## Lecture Notes

By Dr.Subhendu Kumar Rath, BPUT

### Lecture #1

#### Introduction:

The main purpose of parallel computing is to perform computations faster than that can be done with a single processor by using a number of processors concurrently. The need for faster solutions and for solving larger size problems arises in a wide variety of applications. These include fluid dynamics, weather predictions, modeling and simulation of large systems, information processing and extraction, image processing, artificial intelligence and automate manufacturing.

Three main factors have contributed to the current strong trend in favour of parallel processing.

First, the hardware cost has been falling steadily; hence it is now possible to build systems with many processors at a reasonable cost.

Second, the very large scale integration circuit technology has advanced to the point where it is possible to design complex systems requiring millions of transistors on a single chip.

Third, the faster cycle time of a Von-Neumann type processor seem to be approaching fundamental physical limitations beyond which no improvement is possible.

**Parallel Computer:** A parallel Computer is simply a collection of processors, typically of the same type, interconnected in a certain fashion to allow the coordination of their activities and the exchange of data.

**Parallel Computing:** Parallel Computing resembles the study of designing algorithms such that the time complexity is minimum. Thus the speed up factor is taken into consideration.

**Speed up:** Let  $C$  be a computational problem and let  $n$  is the input size.

$T_s(n)$ = Time complexity of best sequential algorithm.

$T_p(n)$ = Time complexity of the parallel algorithm with  $p$  processors.

Then speed up  $S(n,p)=T_s(n)/T_p(n)$

**Note:**  $T_s(n) \leq T_1(n)$ , putting  $n=1$

Since  $T_s(n)$  is the best sequential algorithm.  $T_1(n)$  may not be best.

**Theorem:** If the process is sequential , the speed up cannot exceed the number of processors i.e.,  $S(n,p) \leq p$ , where  $p$  is the number of processors.

**Proof:** We know that  $T_s(n) \leq T_1(n)$ . Suppose it is false for  $p=1$ .

Then  $S(n,1) > 1$

i.e.,  $T_s(n)/T_1(n) > 1$

i.e.,  $T_s(n) > T_1(n)$

but  $T_s(n) \leq T_1(n)$

Hence a contradiction and  $S(n,p) \leq p$  is true.



## Motivating Parallelism:

Development of parallel software has traditionally been thought of as time and effort intensive. This can be largely attributed to the inherent complexity of specifying and coordinating concurrent tasks, a lack of portable algorithms, standardized environments, and software development toolkits. When viewed in the context of the brisk rate of development of microprocessors, one is tempted to question the need for devoting significant effort towards exploiting parallelism as a means of accelerating applications. After all, if it takes two years to develop a parallel application, during which time the underlying hardware and/or software platform has become obsolete, the development effort is clearly wasted.

However, there are some unmistakable trends in hardware design, which indicate that uniprocessor (or implicitly parallel) architectures may not be able to sustain the rate of realizable performance increments in the future. This is a result of lack of implicit parallelism as well as other bottlenecks such as the datapath and the memory. At the same time, standardized hardware interfaces have reduced the turnaround time from the development of a microprocessor to a parallel machine based on the microprocessor.

Furthermore, considerable progress has been made in standardization of programming environments to ensure a longer life-cycle for parallel applications. All of these present compelling arguments in favor of parallel computing platforms.

### **The Computational Power Argument – from Transistors to FLOPS**

In 1965, Gordon Moore made the following simple observation:

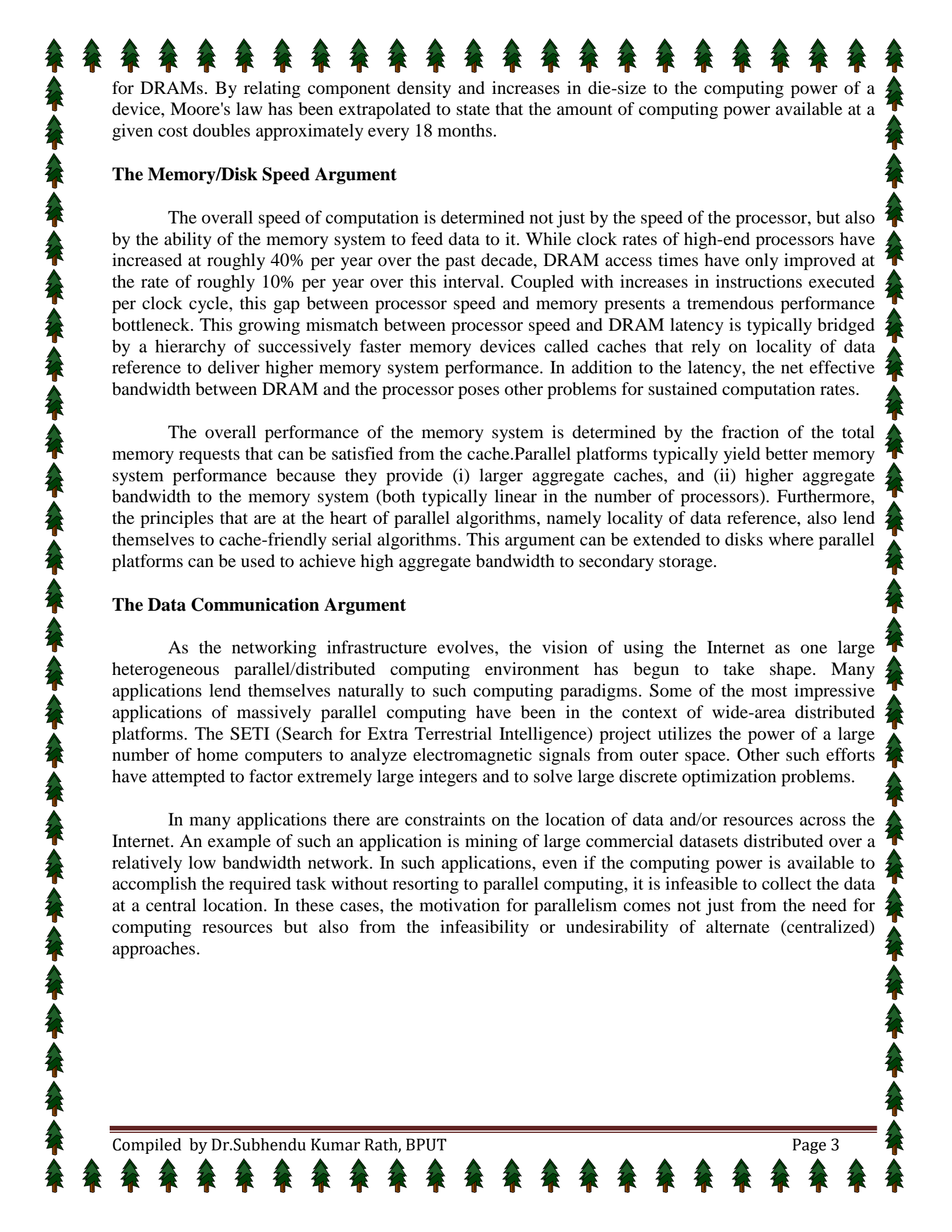
"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."

His reasoning was based on an empirical log-linear relationship between device complexity and time, observed over three data points. He used this to justify that by 1975, devices with as many as 65,000 components would become feasible on a single silicon chip occupying an area of only about one-fourth of a square inch. This projection turned out to be accurate with the fabrication of a 16K CCD memory with about 65,000 components in 1975. In a subsequent paper in 1975, Moore attributed the log-linear relationship to exponential behavior of die sizes, finer minimum dimensions, and "circuit and device cleverness". He went on to state that:

"There is no room left to squeeze anything out by being clever. Going forward from here we have to depend on the two size factors - bigger dies and finer dimensions."

He revised his rate of circuit complexity doubling to 18 months and projected from 1975 onwards at this reduced rate. This curve came to be known as "Moore's Law".

Formally, **Moore's Law** states that circuit complexity doubles every eighteen months. This empirical relationship has been amazingly resilient over the years both for microprocessors as well as



for DRAMs. By relating component density and increases in die-size to the computing power of a device, Moore's law has been extrapolated to state that the amount of computing power available at a given cost doubles approximately every 18 months.

### **The Memory/Disk Speed Argument**

The overall speed of computation is determined not just by the speed of the processor, but also by the ability of the memory system to feed data to it. While clock rates of high-end processors have increased at roughly 40% per year over the past decade, DRAM access times have only improved at the rate of roughly 10% per year over this interval. Coupled with increases in instructions executed per clock cycle, this gap between processor speed and memory presents a tremendous performance bottleneck. This growing mismatch between processor speed and DRAM latency is typically bridged by a hierarchy of successively faster memory devices called caches that rely on locality of data reference to deliver higher memory system performance. In addition to the latency, the net effective bandwidth between DRAM and the processor poses other problems for sustained computation rates.

The overall performance of the memory system is determined by the fraction of the total memory requests that can be satisfied from the cache. Parallel platforms typically yield better memory system performance because they provide (i) larger aggregate caches, and (ii) higher aggregate bandwidth to the memory system (both typically linear in the number of processors). Furthermore, the principles that are at the heart of parallel algorithms, namely locality of data reference, also lend themselves to cache-friendly serial algorithms. This argument can be extended to disks where parallel platforms can be used to achieve high aggregate bandwidth to secondary storage.

### **The Data Communication Argument**

As the networking infrastructure evolves, the vision of using the Internet as one large heterogeneous parallel/distributed computing environment has begun to take shape. Many applications lend themselves naturally to such computing paradigms. Some of the most impressive applications of massively parallel computing have been in the context of wide-area distributed platforms. The SETI (Search for Extra Terrestrial Intelligence) project utilizes the power of a large number of home computers to analyze electromagnetic signals from outer space. Other such efforts have attempted to factor extremely large integers and to solve large discrete optimization problems.

In many applications there are constraints on the location of data and/or resources across the Internet. An example of such an application is mining of large commercial datasets distributed over a relatively low bandwidth network. In such applications, even if the computing power is available to accomplish the required task without resorting to parallel computing, it is infeasible to collect the data at a central location. In these cases, the motivation for parallelism comes not just from the need for computing resources but also from the infeasibility or undesirability of alternate (centralized) approaches.



## Lecture #2

### Scope of Parallel Computing:

Parallel computing has made a tremendous impact on a variety of areas ranging from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing. The cost benefits of parallelism coupled with the performance requirements of applications present compelling arguments in favor of parallel computing. We present a small sample of the diverse applications of parallel computing.

#### **1. Applications in Engineering and Design**

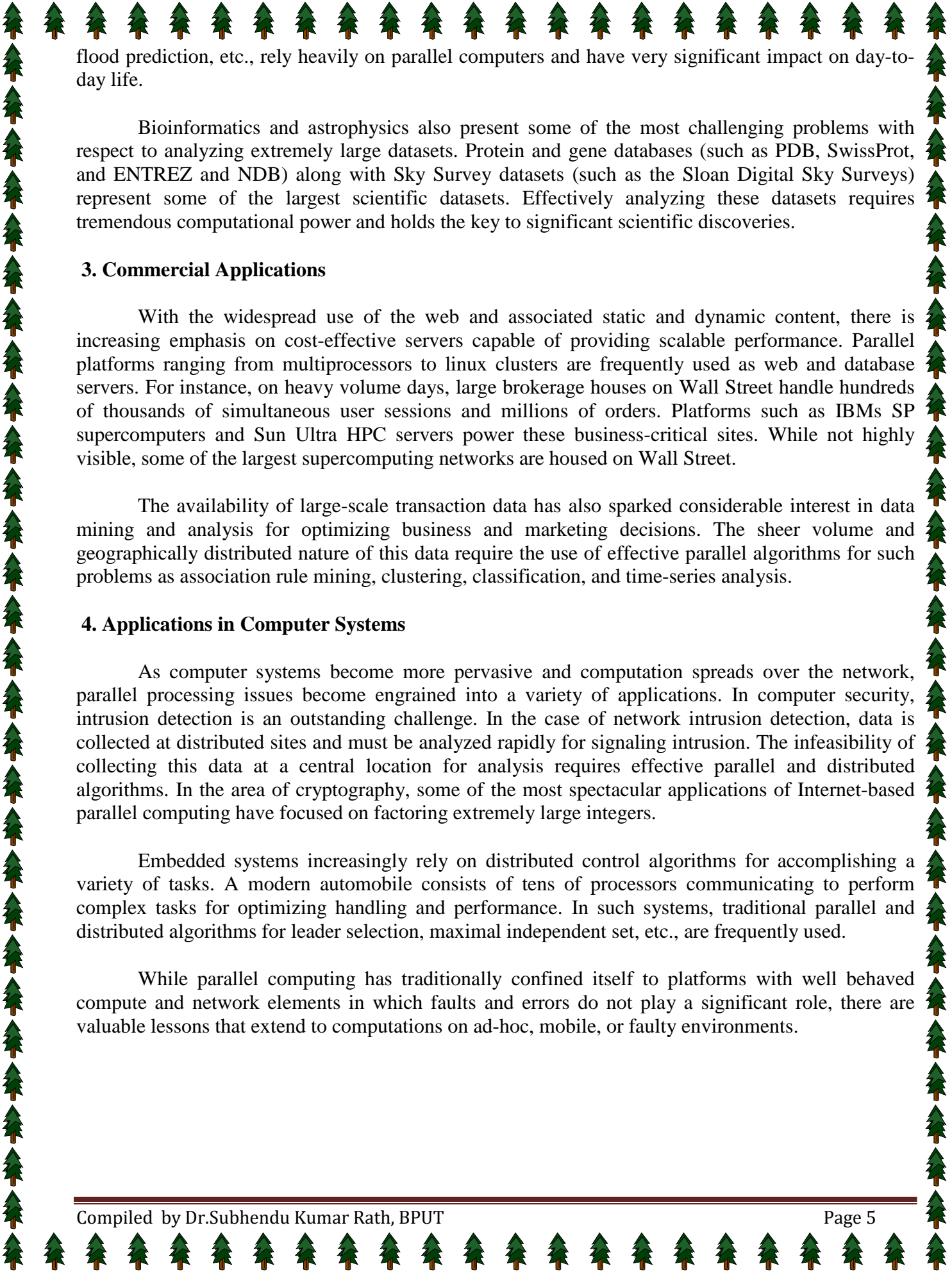
Parallel computing has traditionally been employed with great success in the design of airfoils (optimizing lift, drag, stability), internal combustion engines (optimizing charge distribution, burn), high-speed circuits (layouts for delays and capacitive and inductive effects), and structures (optimizing structural integrity, design parameters, cost, etc.), among others. More recently, design of microelectromechanical and nanoelectromechanical systems (MEMS and NEMS) has attracted significant attention. While most applications in engineering and design pose problems of multiple spatial and temporal scales and coupled physical phenomena, in the case of MEMS/NEMS design these problems are particularly acute. Here, we often deal with a mix of quantum phenomena, molecular dynamics, and stochastic and continuum models with physical processes such as conduction, convection, radiation, and structural mechanics, all in a single system. This presents formidable challenges for geometric modeling, mathematical modeling, and algorithm development, all in the context of parallel computers.

Other applications in engineering and design focus on optimization of a variety of processes. Parallel computers have been used to solve a variety of discrete and continuous optimization problems. Algorithms such as Simplex, Interior Point Method for linear optimization and Branch-and-bound, and Genetic programming for discrete optimization have been efficiently parallelized and are frequently used.

#### **2. Scientific Applications**

The past few years have seen a revolution in high performance scientific computing applications. The sequencing of the human genome by the International Human Genome Sequencing Consortium and Celera, Inc. has opened exciting new frontiers in bioinformatics. Functional and structural characterization of genes and proteins hold the promise of understanding and fundamentally influencing biological processes. Analyzing biological sequences with a view to developing new drugs and cures for diseases and medical conditions requires innovative algorithms as well as large-scale computational power. Indeed, some of the newest parallel computing technologies are targeted specifically towards applications in bioinformatics.

Advances in computational physics and chemistry have focused on understanding processes ranging in scale from quantum phenomena to macromolecular structures. These have resulted in design of new materials, understanding of chemical pathways, and more efficient processes. Applications in astrophysics have explored the evolution of galaxies, thermonuclear processes, and the analysis of extremely large datasets from telescopes. Weather modeling, mineral prospecting,



flood prediction, etc., rely heavily on parallel computers and have very significant impact on day-to-day life.

Bioinformatics and astrophysics also present some of the most challenging problems with respect to analyzing extremely large datasets. Protein and gene databases (such as PDB, SwissProt, and ENTREZ and NDB) along with Sky Survey datasets (such as the Sloan Digital Sky Surveys) represent some of the largest scientific datasets. Effectively analyzing these datasets requires tremendous computational power and holds the key to significant scientific discoveries.

### **3. Commercial Applications**

With the widespread use of the web and associated static and dynamic content, there is increasing emphasis on cost-effective servers capable of providing scalable performance. Parallel platforms ranging from multiprocessors to linux clusters are frequently used as web and database servers. For instance, on heavy volume days, large brokerage houses on Wall Street handle hundreds of thousands of simultaneous user sessions and millions of orders. Platforms such as IBM's SP supercomputers and Sun Ultra HPC servers power these business-critical sites. While not highly visible, some of the largest supercomputing networks are housed on Wall Street.

The availability of large-scale transaction data has also sparked considerable interest in data mining and analysis for optimizing business and marketing decisions. The sheer volume and geographically distributed nature of this data require the use of effective parallel algorithms for such problems as association rule mining, clustering, classification, and time-series analysis.

### **4. Applications in Computer Systems**

As computer systems become more pervasive and computation spreads over the network, parallel processing issues become engrained into a variety of applications. In computer security, intrusion detection is an outstanding challenge. In the case of network intrusion detection, data is collected at distributed sites and must be analyzed rapidly for signaling intrusion. The infeasibility of collecting this data at a central location for analysis requires effective parallel and distributed algorithms. In the area of cryptography, some of the most spectacular applications of Internet-based parallel computing have focused on factoring extremely large integers.

Embedded systems increasingly rely on distributed control algorithms for accomplishing a variety of tasks. A modern automobile consists of tens of processors communicating to perform complex tasks for optimizing handling and performance. In such systems, traditional parallel and distributed algorithms for leader selection, maximal independent set, etc., are frequently used.

While parallel computing has traditionally confined itself to platforms with well behaved compute and network elements in which faults and errors do not play a significant role, there are valuable lessons that extend to computations on ad-hoc, mobile, or faulty environments.



## Lecture #3

### Parallel Programming Platforms:

The traditional logical view of a sequential computer consists of a memory connected to a processor via a datapath. All three components – processor, memory, and datapath – present bottlenecks to the overall processing rate of a computer system. A number of architectural innovations over the years have addressed these bottlenecks. One of the most important innovations is multiplicity – in processing units, datapaths, and memory units. This multiplicity is either entirely hidden from the programmer, as in the case of implicit parallelism, or exposed to the programmer in different forms. In this chapter, we present an overview of important architectural concepts as they relate to parallel processing. The objective is to provide sufficient detail for programmers to be able to write efficient code on a variety of platforms. We develop cost models and abstractions for quantifying the performance of various parallel algorithms, and identify bottlenecks resulting from various programming constructs.

### **Pipelining**

Processors have long relied on pipelines for improving execution rates. By overlapping various stages in instruction execution (fetch, schedule, decode, operand fetch, execute, store, among others), pipelining enables faster execution.

**Pipelining** is a technique of dividing one task into multiple subtasks and executing the subtasks in parallel with multiple hardware units.

In a pipelining processor, multiple instructions are executed at various stages of instruction cycle simultaneously in different sections of the processor.

Ex: In pipelining processors,  $S_1, S_2, S_3, \dots, S_n$  are  $n$  sections that form the pipeline. Each section performs a subtask on the input received from the inter-stage buffer that stores the output of the previous section. All sections can perform their operations simultaneously.

The objective of pipelining processing is to increase throughput due to the overlap between the execution of the consecutive task.

There are two types of pipelines:

1. Instruction Pipeline
2. Arithmetic pipeline

#### **1. Instruction pipeline**

An instruction pipeline divides an instruction cycle actions into multiple steps that are executed one-by-one in different sections of the processor.

The number of sections in the pipeline is designed by the computer architect.



Consider the instruction cycle is divided into four steps:

- a) Instruction Fetch(IF)
- b) Instruction Decode(ID)
- c) Execute(EX)
- d) Write Result(WR)

Assume that there are  $m$  instructions and these instructions will be executed in  $n$ -sections of the pipeline processor.

The time taken for the first instruction =  $nt_c$ , where  $t_c$  is the duration of the clock cycle.

Time taken for remaining  $(m-1)$  instructions =  $(m-1)t_c$

Total time taken for  $m$  instructions =  $nt_c + (m-1)t_c = (n+m-1)t_c$

If the processor is non-pipelined, then the total time taken for  $m$  instructions is  $mnt_c$ .

Hence performance gain due to pipeline =  $mnt_c / (m+n-1)t_c$

The assembly-line analogy works well for understanding pipelines. If the assembly of a car, taking 100 time units, can be broken into 10 pipelined stages of 10 units each, a single assembly line can produce a car every 10 time units! This represents a 10-fold speedup over producing cars entirely serially, one after the other. It is also evident from this example that to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units, thus lengthening the pipeline and increasing overlap in execution. In the context of processors, this enables faster clock rates since the tasks are now smaller.

For example, the Pentium 4, which operates at 2.0 GHz, has a 20 stage pipeline. Note that the speed of a single pipeline is ultimately limited by the largest atomic task in the pipeline. Furthermore, in typical instruction traces, every fifth to sixth instruction is a branch instruction. Long instruction pipelines therefore need effective techniques for predicting branch destinations so that pipelines can be speculatively filled. The penalty of a misprediction increases as the pipelines become deeper since a larger number of instructions need to be flushed. These factors place limitations on the depth of a processor pipeline and the resulting performance gains.

An obvious way to improve instruction execution rate beyond this level is to use multiple pipelines. During each clock cycle, multiple instructions are piped into the processor in parallel. These instructions are executed on multiple functional units.



## Lecture #4

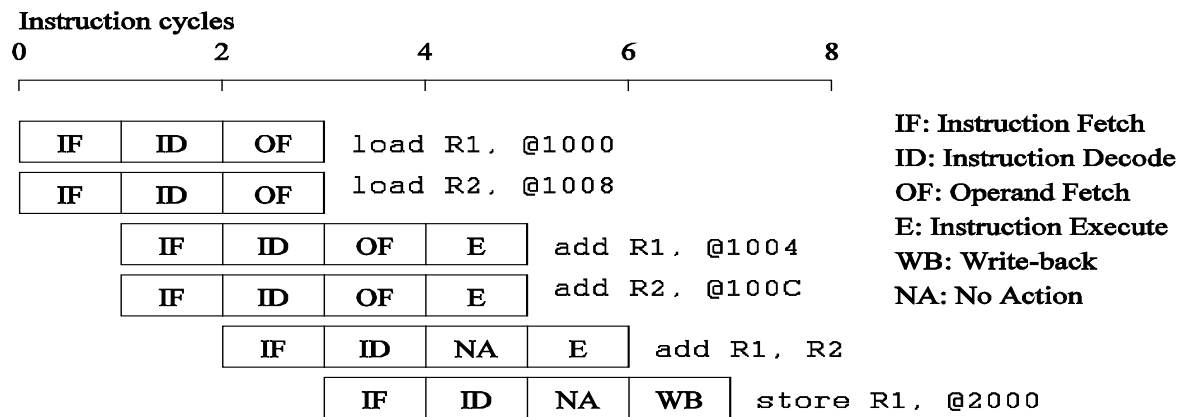
### Superscalar Execution:

Consider a processor with two pipelines and the ability to simultaneously issue two instructions. These processors are sometimes also referred to as **super-pipelined processors**. The ability of a processor to issue multiple instructions in the same cycle is referred to as **superscalar execution**. Since the architecture illustrated in the following figure allows two issues per clock cycle, it is also referred to as two-way superscalar or dual issue execution.

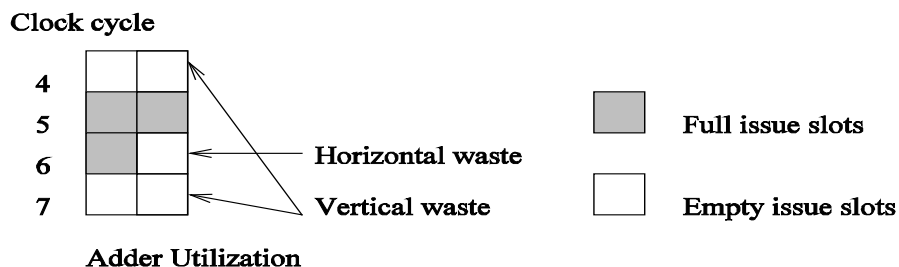
Figure : Example of a two-way superscalar execution of instructions.

<ol style="list-style-type: none"> <li>1. load R1, @1000</li> <li>2. load R2, @1008</li> <li>3. add R1, @1004</li> <li>4. add R2, @100C</li> <li>5. add R1, R2</li> <li>6. store R1, @2000</li> </ol>	<ol style="list-style-type: none"> <li>1. load R1, @1000</li> <li>2. add R1, @1004</li> <li>3. add R1, @1008</li> <li>4. add R1, @100C</li> <li>5. store R1, @2000</li> </ol>	<ol style="list-style-type: none"> <li>1. load R1, @1000</li> <li>2. add R1, @1004</li> <li>3. load R2, @1008</li> <li>4. add R2, @100C</li> <li>5. add R1, R2</li> <li>6. store R1, @2000</li> </ol>
(i)	(ii)	(iii)

(a) Three different code fragments for adding a list of four numbers.

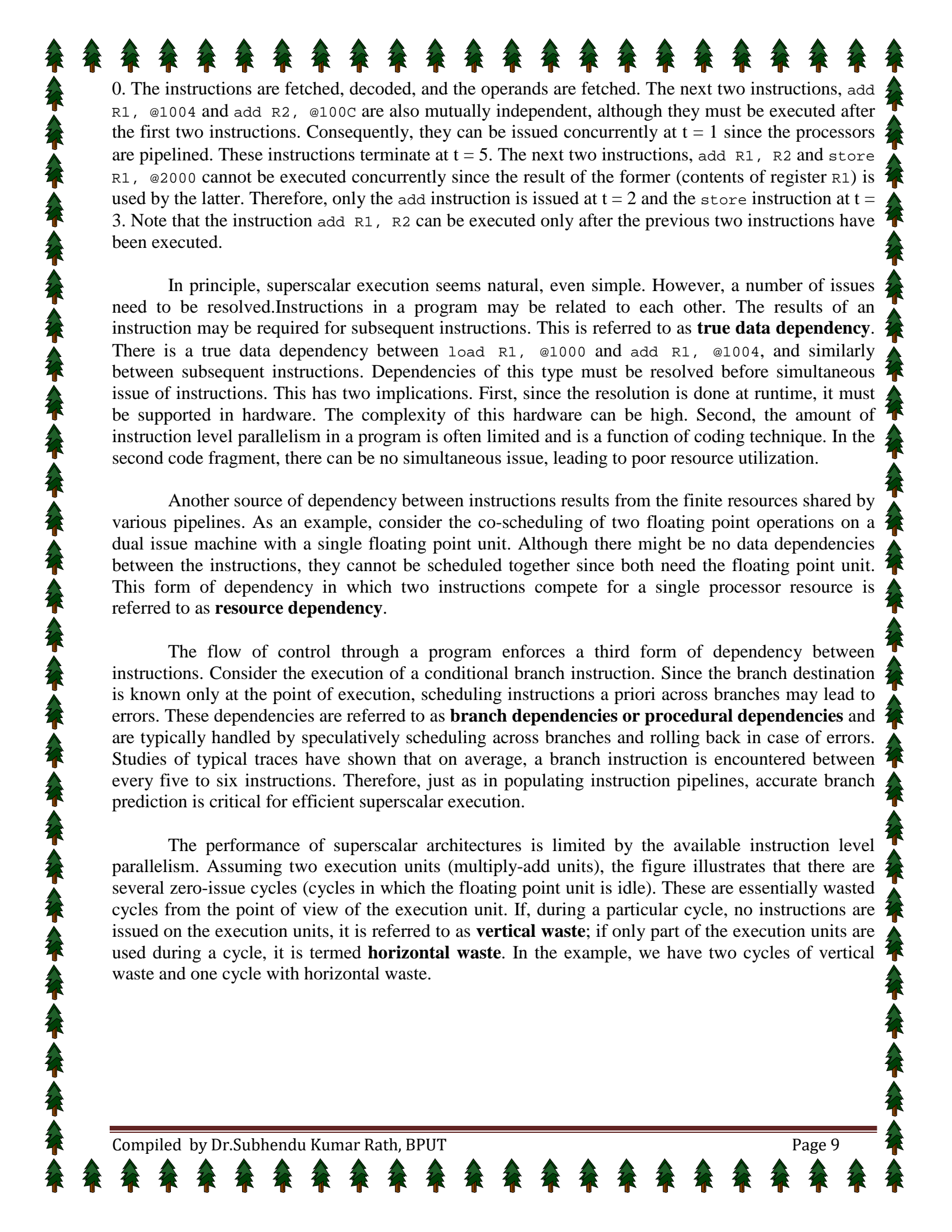


(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

Consider the execution of the first code fragment in the figure for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently. This is illustrated in the simultaneous issue of the instructions `load R1, @1000` and `load R2, @1008` at  $t =$



0. The instructions are fetched, decoded, and the operands are fetched. The next two instructions, `add R1, @1004` and `add R2, @100C` are also mutually independent, although they must be executed after the first two instructions. Consequently, they can be issued concurrently at  $t = 1$  since the processors are pipelined. These instructions terminate at  $t = 5$ . The next two instructions, `add R1, R2` and `store R1, @2000` cannot be executed concurrently since the result of the former (contents of register  $R1$ ) is used by the latter. Therefore, only the `add` instruction is issued at  $t = 2$  and the `store` instruction at  $t = 3$ . Note that the instruction `add R1, R2` can be executed only after the previous two instructions have been executed.

In principle, superscalar execution seems natural, even simple. However, a number of issues need to be resolved. Instructions in a program may be related to each other. The results of an instruction may be required for subsequent instructions. This is referred to as **true data dependency**. There is a true data dependency between `load R1, @1000` and `add R1, @1004`, and similarly between subsequent instructions. Dependencies of this type must be resolved before simultaneous issue of instructions. This has two implications. First, since the resolution is done at runtime, it must be supported in hardware. The complexity of this hardware can be high. Second, the amount of instruction level parallelism in a program is often limited and is a function of coding technique. In the second code fragment, there can be no simultaneous issue, leading to poor resource utilization.

Another source of dependency between instructions results from the finite resources shared by various pipelines. As an example, consider the co-scheduling of two floating point operations on a dual issue machine with a single floating point unit. Although there might be no data dependencies between the instructions, they cannot be scheduled together since both need the floating point unit. This form of dependency in which two instructions compete for a single processor resource is referred to as **resource dependency**.

The flow of control through a program enforces a third form of dependency between instructions. Consider the execution of a conditional branch instruction. Since the branch destination is known only at the point of execution, scheduling instructions a priori across branches may lead to errors. These dependencies are referred to as **branch dependencies or procedural dependencies** and are typically handled by speculatively scheduling across branches and rolling back in case of errors. Studies of typical traces have shown that on average, a branch instruction is encountered between every five to six instructions. Therefore, just as in populating instruction pipelines, accurate branch prediction is critical for efficient superscalar execution.

The performance of superscalar architectures is limited by the available instruction level parallelism. Assuming two execution units (multiply-add units), the figure illustrates that there are several zero-issue cycles (cycles in which the floating point unit is idle). These are essentially wasted cycles from the point of view of the execution unit. If, during a particular cycle, no instructions are issued on the execution units, it is referred to as **vertical waste**; if only part of the execution units are used during a cycle, it is termed **horizontal waste**. In the example, we have two cycles of vertical waste and one cycle with horizontal waste.



## Lecture #5

### **Very Long Instruction Word(VLIW) Processors:**

The parallelism extracted by superscalar processors is often limited by the instruction look-ahead. The hardware logic for dynamic dependency analysis is typically in the range of 5-10% of the total logic on conventional microprocessors (about 5% on the four-way superscalar Sun UltraSPARC). This complexity grows roughly quadratically with the number of issues and can become a bottleneck.

An alternate concept for exploiting instruction-level parallelism used in **very long instruction word** (VLIW) processors relies on the compiler to resolve dependencies and resource availability at compile time. Instructions that can be executed concurrently are packed into groups and parceled off to the processor as a single long instruction word (thus the name) to be executed on multiple functional units at the same time.

The VLIW concept has both advantages and disadvantages compared to superscalar processors. Since scheduling is done in software, the decoding and instruction issue mechanisms are simpler in VLIW processors. The compiler has a larger context from which to select instructions and can use a variety of transformations to optimize parallelism when compared to a hardware issue unit. Additional parallel instructions are typically made available to the compiler to control parallel execution. However, compilers do not have the dynamic program state (e.g., the branch history buffer) available to make scheduling decisions. This reduces the accuracy of branch and memory prediction, but allows the use of more sophisticated static prediction schemes. Other runtime situations such as stalls on data fetch because of cache misses are extremely difficult to predict accurately. This limits the scope and performance of static compiler-based scheduling.

Finally, the performance of VLIW processors is very sensitive to the compilers' ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism. Loop unrolling, branch prediction and speculative execution all play important roles in the performance of VLIW processors. While superscalar and VLIW processors have been successful in exploiting implicit parallelism, they are generally limited to smaller scales of concurrency in the range of four- to eight-way parallelism.

### **Limitations of Memory System Performance\***

The effective performance of a program on a computer relies not just on the speed of the processor but also on the ability of the memory system to feed data to the processor. At the logical level, a memory system, possibly consisting of multiple levels of caches, takes in a request for a memory word and returns a block of data of size  $b$  containing the requested word after  $l$  nanoseconds. Here,  $l$  is referred to as the **latency** of the memory. The rate at which data can be pumped from the memory to the processor determines the **bandwidth** of the memory system.

To study the effect of memory system latency, we assume in the following examples that a memory block consists of one word. We later relax this assumption while examining the role of memory bandwidth. Since we are primarily interested in maximum achievable performance, we also assume the best case cache-replacement policy. We refer the reader to the bibliography for a detailed discussion of memory system design.



## Example : Effect of memory latency on performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The peak processor rating is therefore 4 GFLOPS. Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. Consider the problem of computing the dot-product of two vectors on such a platform. A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch. It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating. This example highlights the need for effective memory system performance in achieving high computation rates.

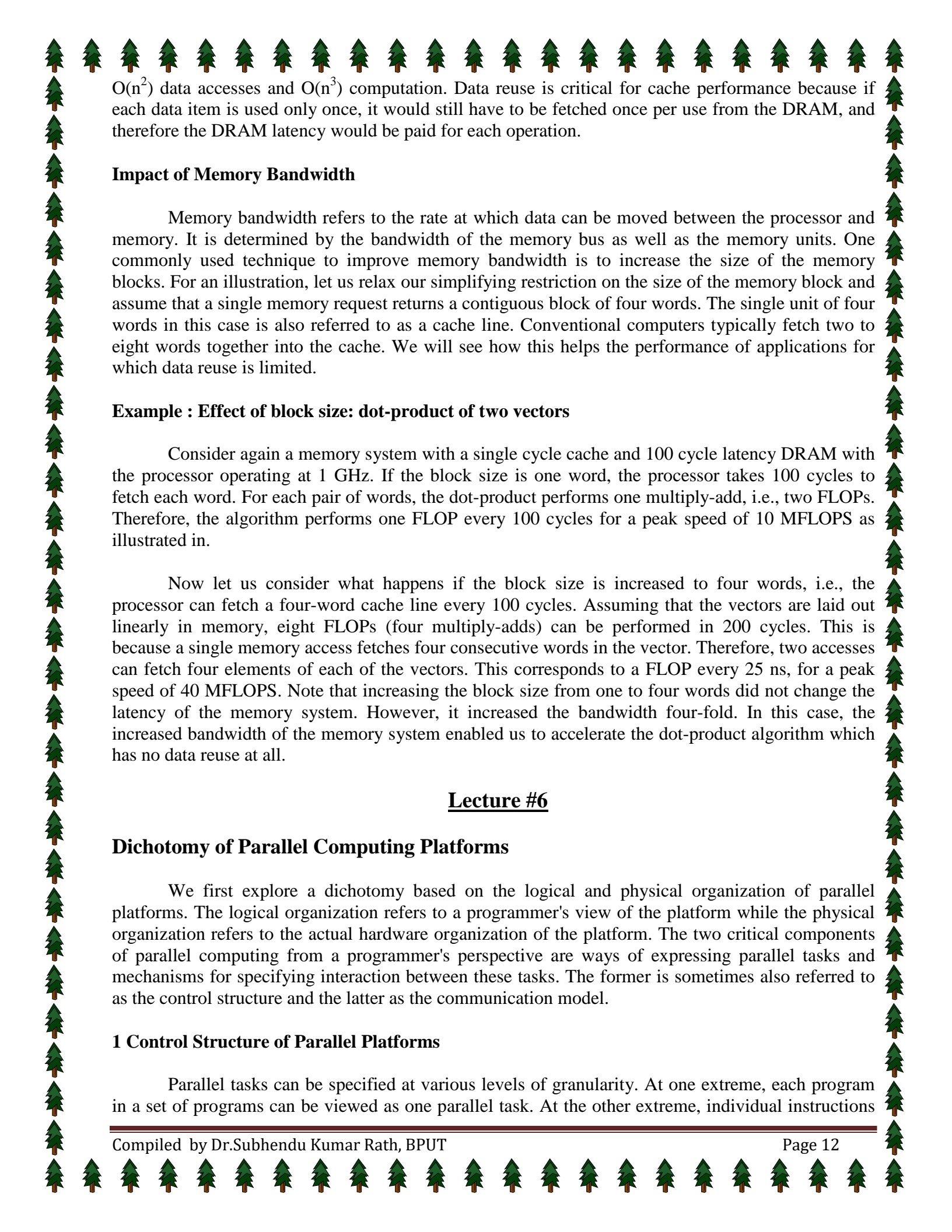
### Improving Effective Memory Latency Using Caches

Handling the mismatch in processor and DRAM speeds has motivated a number of architectural innovations in memory system design. One such innovation addresses the speed mismatch by placing a smaller and faster memory between the processor and the DRAM. This memory, referred to as the cache, acts as a low-latency high-bandwidth storage. The data needed by the processor is first fetched into the cache. All subsequent accesses to data items residing in the cache are serviced by the cache. Thus, in principle, if a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache. The fraction of data references satisfied by the cache is called the cache hit ratio of the computation on the system. The effective computation rate of many applications is bounded not by the processing rate of the CPU, but by the rate at which data can be pumped into the CPU. Such computations are referred to as being memory bound. The performance of memory bound programs is critically impacted by the cache hit ratio.

### Example: Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor with a 100 ns latency DRAM. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle (typically on the processor itself). We use this setup to multiply two matrices A and B of dimensions 32 x 32. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C. Once again, we assume an ideal cache placement strategy in which none of the data items are overwritten by others. Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200  $\mu$ s. We know from elementary algorithmics that multiplying two  $n \times n$  matrices takes  $2n^3$  operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16  $\mu$ s) at four instructions per cycle. The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200+16  $\mu$ s. This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS. Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably.

The improvement in performance resulting from the presence of the cache is based on the assumption that there is repeated reference to the same data item. This notion of repeated reference to a data item in a small time window is called temporal locality of reference. In our example, we had



$O(n^2)$  data accesses and  $O(n^3)$  computation. Data reuse is critical for cache performance because if each data item is used only once, it would still have to be fetched once per use from the DRAM, and therefore the DRAM latency would be paid for each operation.

### **Impact of Memory Bandwidth**

Memory bandwidth refers to the rate at which data can be moved between the processor and memory. It is determined by the bandwidth of the memory bus as well as the memory units. One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks. For an illustration, let us relax our simplifying restriction on the size of the memory block and assume that a single memory request returns a contiguous block of four words. The single unit of four words in this case is also referred to as a cache line. Conventional computers typically fetch two to eight words together into the cache. We will see how this helps the performance of applications for which data reuse is limited.

#### **Example : Effect of block size: dot-product of two vectors**

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in.

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

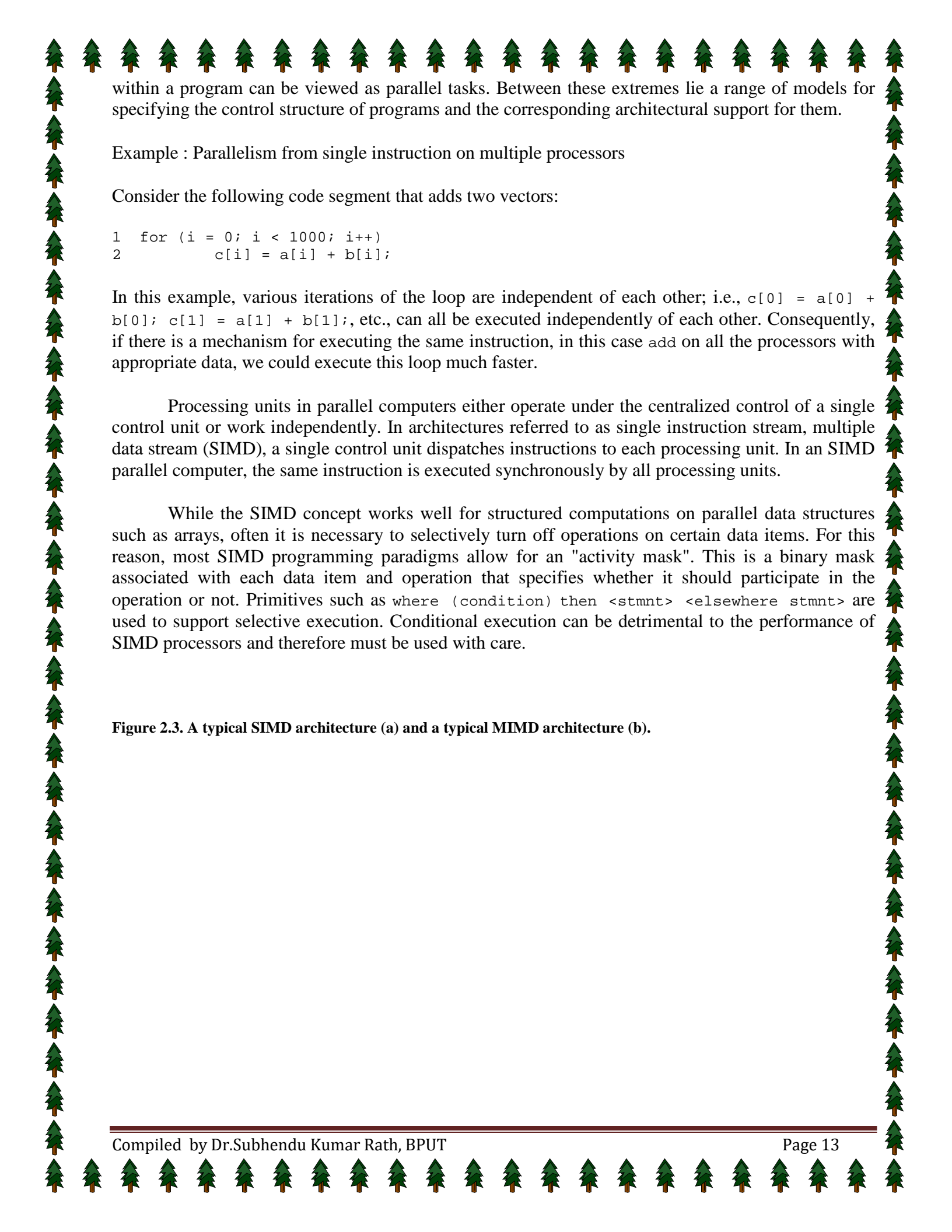
## **Lecture #6**

### **Dichotomy of Parallel Computing Platforms**

We first explore a dichotomy based on the logical and physical organization of parallel platforms. The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks and mechanisms for specifying interaction between these tasks. The former is sometimes also referred to as the control structure and the latter as the communication model.

#### **1 Control Structure of Parallel Platforms**

Parallel tasks can be specified at various levels of granularity. At one extreme, each program in a set of programs can be viewed as one parallel task. At the other extreme, individual instructions



within a program can be viewed as parallel tasks. Between these extremes lie a range of models for specifying the control structure of programs and the corresponding architectural support for them.

Example : Parallelism from single instruction on multiple processors

Consider the following code segment that adds two vectors:

```
1 for (i = 0; i < 1000; i++)  
2     c[i] = a[i] + b[i];
```

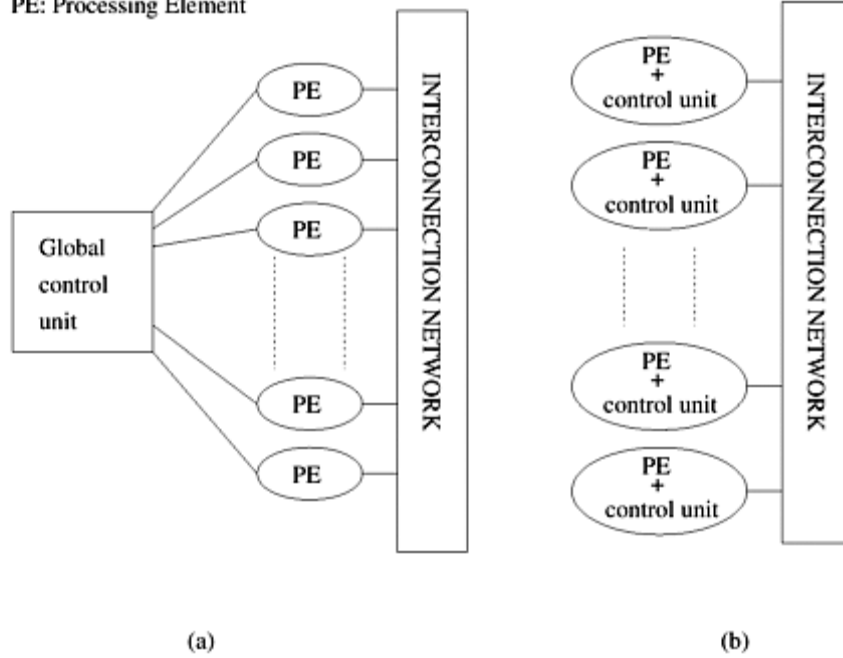
In this example, various iterations of the loop are independent of each other; i.e.,  $c[0] = a[0] + b[0]$ ;  $c[1] = a[1] + b[1]$ ; etc., can all be executed independently of each other. Consequently, if there is a mechanism for executing the same instruction, in this case `add` on all the processors with appropriate data, we could execute this loop much faster.

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as single instruction stream, multiple data stream (SIMD), a single control unit dispatches instructions to each processing unit. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units.

While the SIMD concept works well for structured computations on parallel data structures such as arrays, often it is necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an "activity mask". This is a binary mask associated with each data item and operation that specifies whether it should participate in the operation or not. Primitives such as `where (condition) then <stmt> <elsewhere stmt>` are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors and therefore must be used with care.

Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).

PE: Processing Element



In contrast to SIMD architectures, computers in which each processing element is capable of executing a different program independent of the other processing elements are called multiple instruction stream, multiple data stream (MIMD) computers. Figure (b) depicts a typical MIMD computer. A simple variant of this model, called the single program multiple data (SPMD) model, relies on multiple instances of the same program executing on different data. It is easy to see that the SPMD model has the same expressiveness as the MIMD model since each of the multiple programs can be inserted into one large `if-else` block with conditions specified by the task identifiers. The SPMD model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures, economic factors, design constraints, product life-cycle, and application characteristics. In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time. SIMD computers require extensive design effort resulting in longer product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. The irregular nature of many applications also makes SIMD architectures less suitable.

## Lecture #7

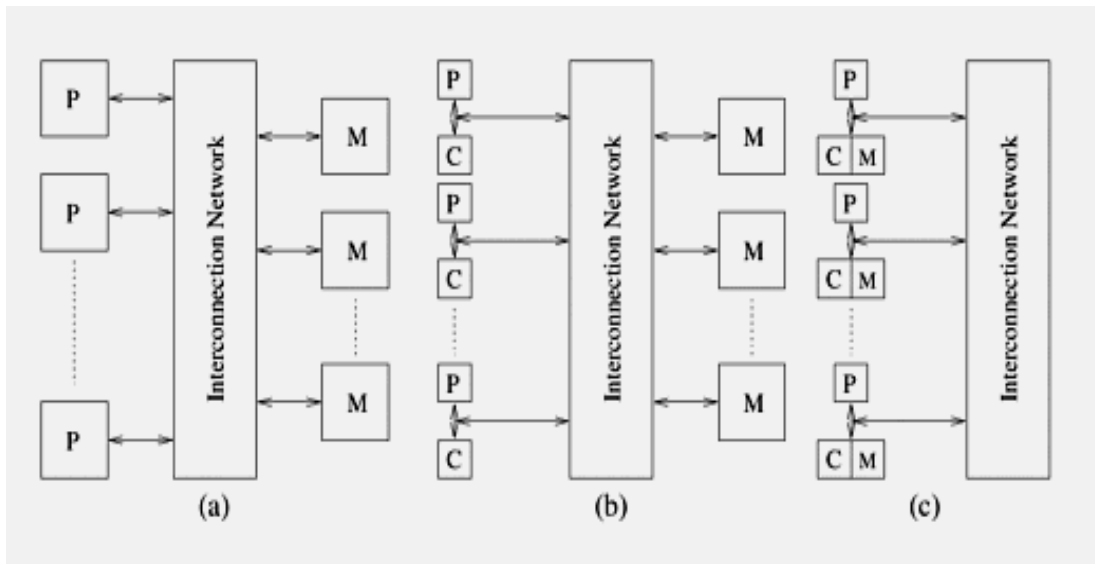
### Communication Model of Parallel Platforms

There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

#### Shared-Address-Space Platforms

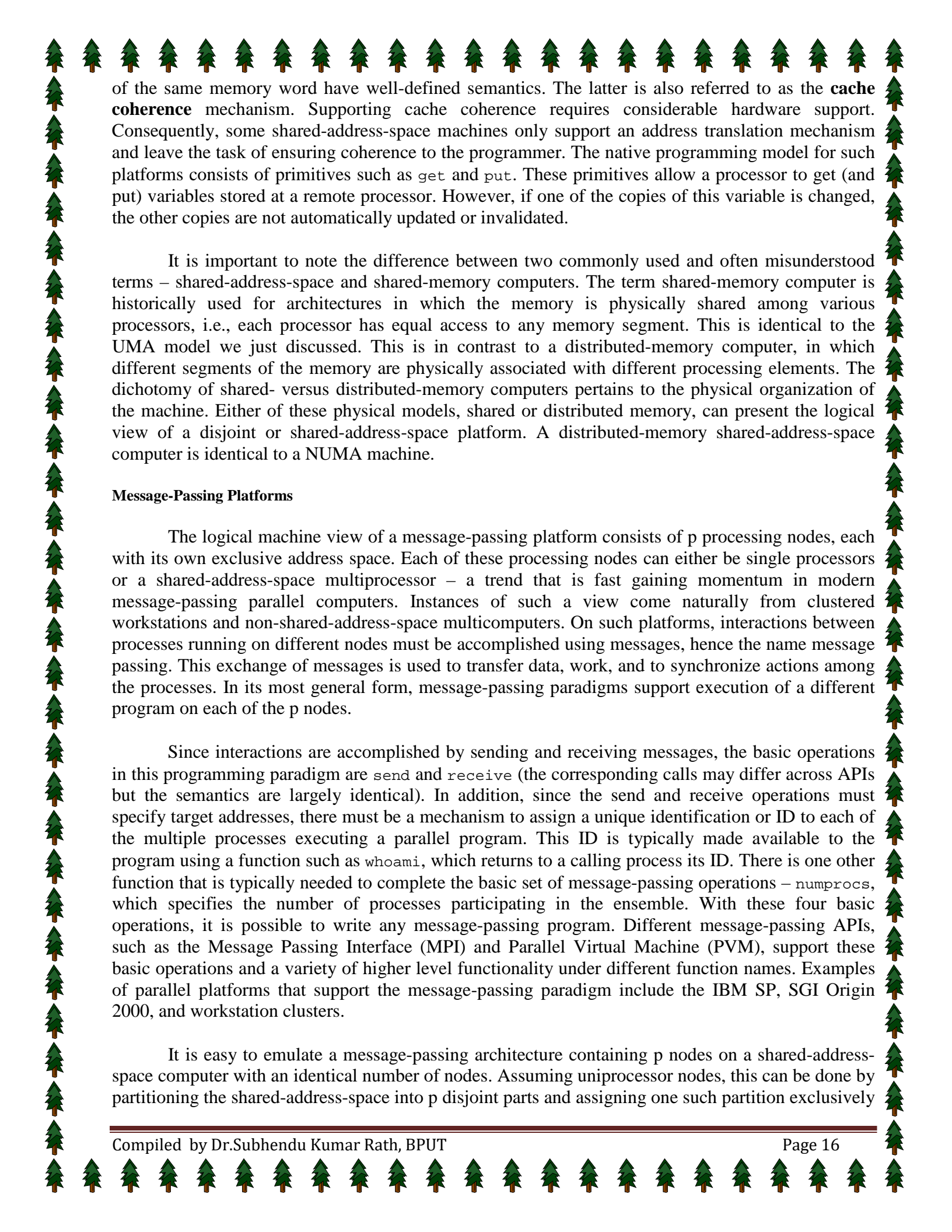
The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. Shared-address-space platforms supporting SPMD programming are also referred to as multiprocessors. Memory in shared-address-space platforms can be local (exclusive to a processor) or global (common to all processors). If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a uniform memory access (UMA) multicomputer. On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a non-uniform memory access (NUMA) multicomputer. Consequently, even a uniprocessor would not be termed UMA if cache access times are considered. For this reason, we define NUMA and UMA architectures only in terms of memory access times and not cache access times. Machines such as the SGI Origin 2000 and Sun Ultra HPC servers belong to the class of NUMA multiprocessors. The distinction between UMA and NUMA platforms is important. If accessing local memory is cheaper than accessing global memory, algorithms must build locality and structure data and computation accordingly.

Figure . Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.



The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time. Supporting a shared-address-space in this context involves two major tasks: providing an address translation mechanism that locates a memory word in the system, and ensuring that concurrent operations on multiple copies





of the same memory word have well-defined semantics. The latter is also referred to as the **cache coherence** mechanism. Supporting cache coherence requires considerable hardware support. Consequently, some shared-address-space machines only support an address translation mechanism and leave the task of ensuring coherence to the programmer. The native programming model for such platforms consists of primitives such as `get` and `put`. These primitives allow a processor to get (and put) variables stored at a remote processor. However, if one of the copies of this variable is changed, the other copies are not automatically updated or invalidated.

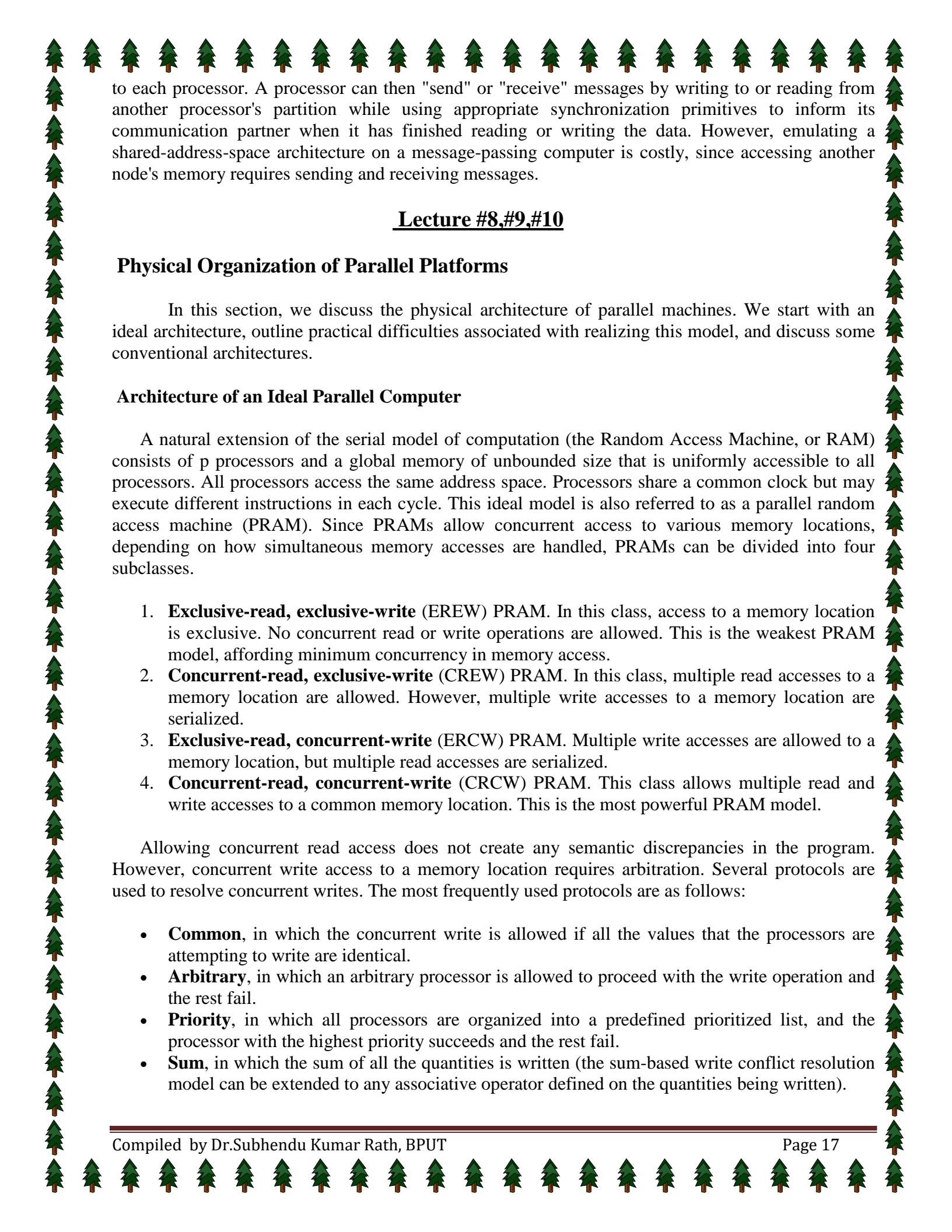
It is important to note the difference between two commonly used and often misunderstood terms – shared-address-space and shared-memory computers. The term shared-memory computer is historically used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment. This is identical to the UMA model we just discussed. This is in contrast to a distributed-memory computer, in which different segments of the memory are physically associated with different processing elements. The dichotomy of shared- versus distributed-memory computers pertains to the physical organization of the machine. Either of these physical models, shared or distributed memory, can present the logical view of a disjoint or shared-address-space platform. A distributed-memory shared-address-space computer is identical to a NUMA machine.

### 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.

## Lecture #8,#9,#10

### 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.

#### 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).

## Architectural Complexity of the Ideal Model

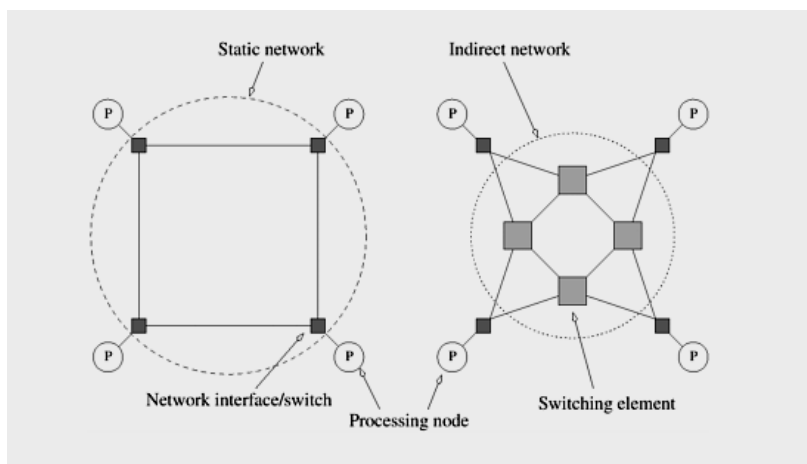
Consider the implementation of an EREW PRAM as a shared-memory computer with  $p$  processors and a global memory of  $m$  words. The processors are connected to the memory through a set of switches. These switches determine the memory word being accessed by each processor. In an EREW PRAM, each of the  $p$  processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously. To ensure such connectivity, the total number of switches must be  $\Theta(mp)$ . For a reasonable memory size, constructing a switching network of this complexity is very expensive. Thus, PRAM models of computation are impossible to realize in practice.

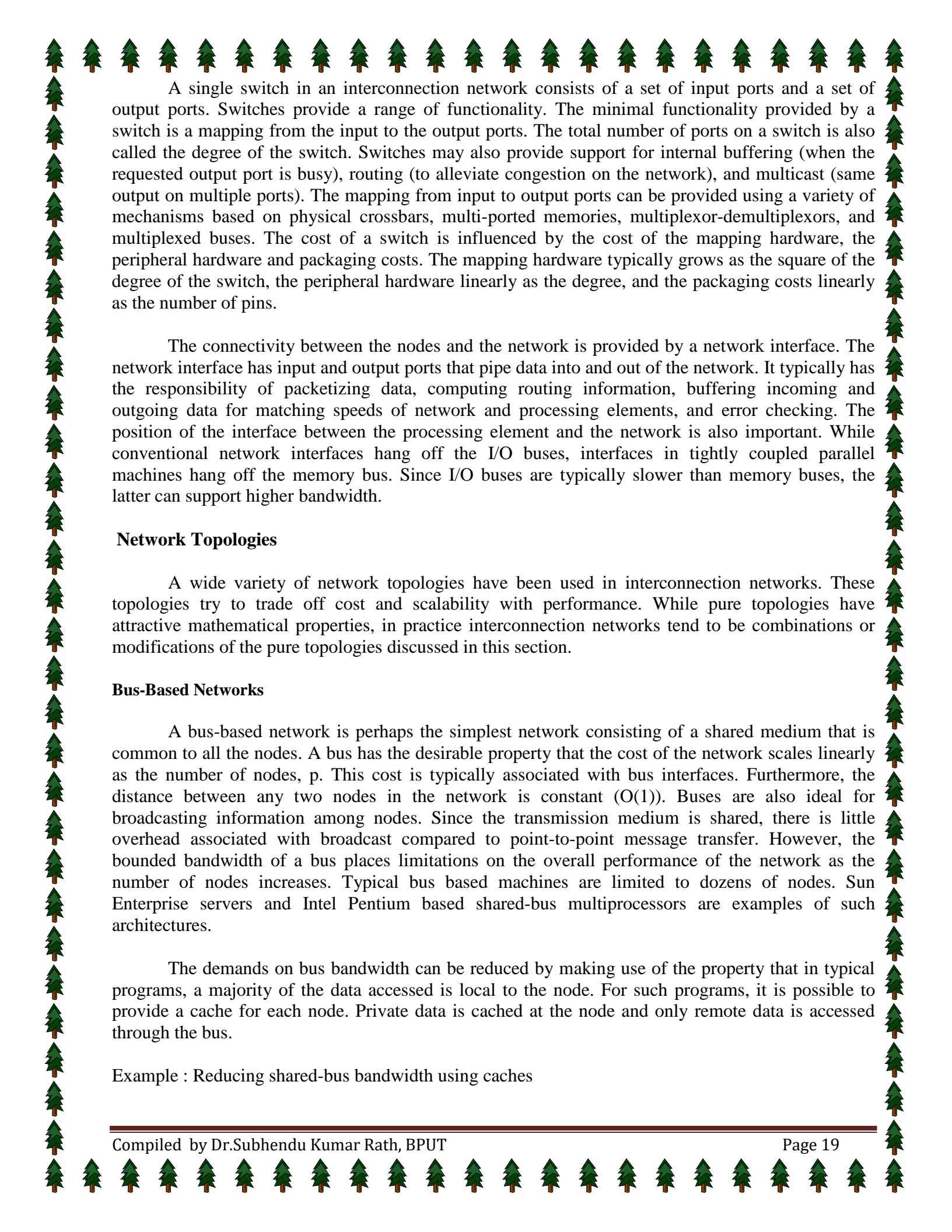
## Interconnection Networks for Parallel Computers

Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules. A black box view of an interconnection network consists of  $n$  inputs and  $m$  outputs. The outputs may or may not be distinct from the inputs. Typical interconnection networks are built using links and switches. A link corresponds to physical media such as a set of wires or fibers capable of carrying information. A variety of factors influence link characteristics. For links based on conducting media, the capacitive coupling between wires limits the speed of signal propagation. This capacitive coupling and attenuation of signal strength are functions of the length of the link.

Interconnection networks can be classified as static or dynamic. Static networks consist of point-to-point communication links among processing nodes and are also referred to as direct networks. Dynamic networks, on the other hand, are built using switches and communication links. Communication links are connected to one another dynamically by the switches to establish paths among processing nodes and memory banks. Dynamic networks are also referred to as indirect networks. Figure (a) illustrates a simple static network of four processing elements or nodes. Each processing node is connected via a network interface to two other nodes in a mesh configuration. Figure (b) illustrates a dynamic network of four nodes connected via a network of switches to other nodes.

Figure . Classification of interconnection networks: (a) a static network; and (b) a dynamic network.





A single switch in an interconnection network consists of a set of input ports and a set of output ports. Switches provide a range of functionality. The minimal functionality provided by a switch is a mapping from the input to the output ports. The total number of ports on a switch is also called the degree of the switch. Switches may also provide support for internal buffering (when the requested output port is busy), routing (to alleviate congestion on the network), and multicast (same output on multiple ports). The mapping from input to output ports can be provided using a variety of mechanisms based on physical crossbars, multi-ported memories, multiplexor-demultiplexors, and multiplexed buses. The cost of a switch is influenced by the cost of the mapping hardware, the peripheral hardware and packaging costs. The mapping hardware typically grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

The connectivity between the nodes and the network is provided by a network interface. The network interface has input and output ports that pipe data into and out of the network. It typically has the responsibility of packetizing data, computing routing information, buffering incoming and outgoing data for matching speeds of network and processing elements, and error checking. The position of the interface between the processing element and the network is also important. While conventional network interfaces hang off the I/O buses, interfaces in tightly coupled parallel machines hang off the memory bus. Since I/O buses are typically slower than memory buses, the latter can support higher bandwidth.

### **Network Topologies**

A wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. While pure topologies have attractive mathematical properties, in practice interconnection networks tend to be combinations or modifications of the pure topologies discussed in this section.

### **Bus-Based Networks**

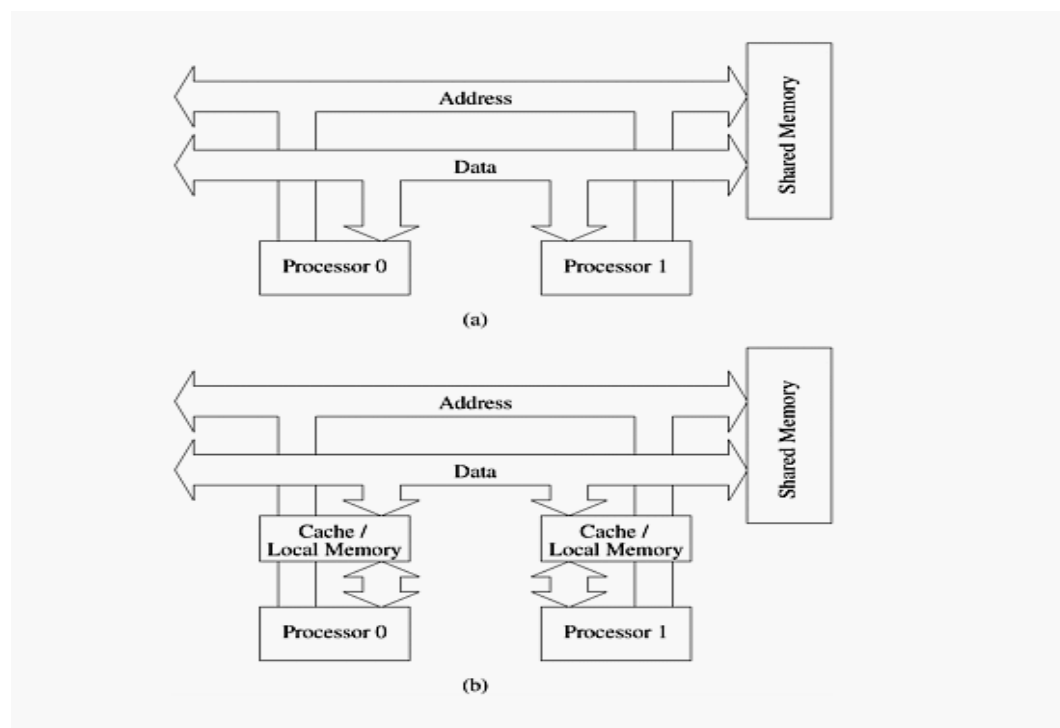
A bus-based network is perhaps the simplest network consisting of a shared medium that is common to all the nodes. A bus has the desirable property that the cost of the network scales linearly as the number of nodes,  $p$ . This cost is typically associated with bus interfaces. Furthermore, the distance between any two nodes in the network is constant ( $O(1)$ ). Buses are also ideal for broadcasting information among nodes. Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer. However, the bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is local to the node. For such programs, it is possible to provide a cache for each node. Private data is cached at the node and only remote data is accessed through the bus.

Example : Reducing shared-bus bandwidth using caches

Figure (a) illustrates  $p$  processors sharing a bus to the memory. Assuming that each processor accesses  $k$  data items, and each data access takes time  $t_{\text{cycle}}$ , the execution time is lower bounded by  $t_{\text{cycle}} \times kp$  seconds. Now consider the hardware organization of Figure (b). Let us assume that 50% of the memory accesses ( $0.5k$ ) are made to local data. This local data resides in the private memory of the processor. We assume that access time to the private memory is identical to the global memory, i.e.,  $t_{\text{cycle}}$ . In this case, the total execution time is lower bounded by  $0.5 \times t_{\text{cycle}} \times k + 0.5 \times t_{\text{cycle}} \times kp$ . Here, the first term results from accesses to local data and the second term from access to shared data. It is easy to see that as  $p$  becomes large, the organization of Figure (b) results in a lower bound that approaches  $0.5 \times t_{\text{cycle}} \times kp$ . This time is a 50% improvement in lower bound on execution time compared to the organization of Figure (a).

**Figure** Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

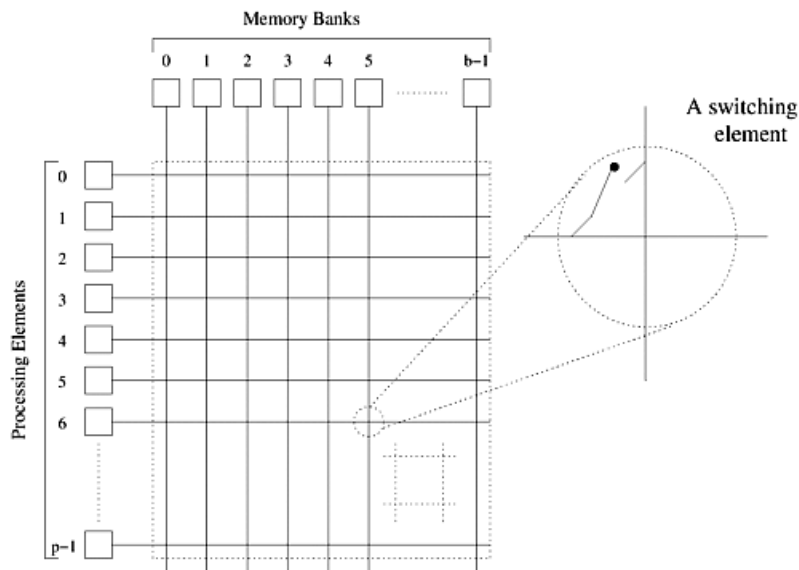


In practice, shared and private data is handled in a more sophisticated manner.

### Crossbar Networks

A simple way to connect  $p$  processors to  $b$  memory banks is to use a crossbar network. A crossbar network employs a grid of switches or switching nodes as shown in the figure. The crossbar network is a non-blocking network in the sense that the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

**Figure .** A completely non-blocking crossbar network connecting  $p$  processors to  $b$  memory banks.



The total number of switching nodes required to implement such a network is  $\Theta(pb)$ . It is reasonable to assume that the number of memory banks  $b$  is at least  $p$ ; otherwise, at any given time, there will be some processing nodes that will be unable to access any memory banks. Therefore, as the value of  $p$  is increased, the complexity (component count) of the switching network grows as  $\Omega(p^2)$ . (See the Appendix for an explanation of the  $\Omega$  notation.) As the number of processing nodes becomes large, this switch complexity is difficult to realize at high data rates. Consequently, crossbar networks are not very scalable in terms of cost.

### Multistage Networks

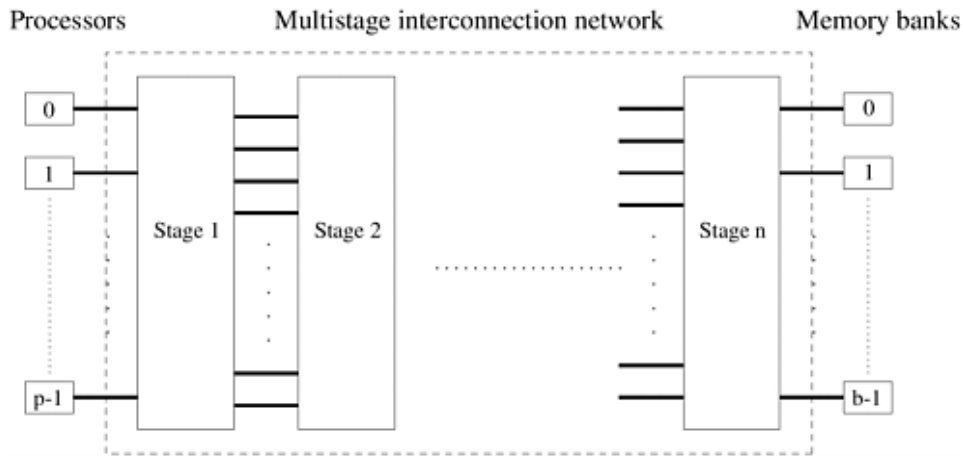
The crossbar interconnection network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. An intermediate class of networks called multistage interconnection networks lies between these two extremes. It is more scalable than the bus in terms of performance and more scalable than the crossbar in terms of cost.

The general schematic of a multistage network consisting of  $p$  processing nodes and  $b$  memory banks is shown in the figure. A commonly used multistage connection network is the omega network. This network consists of  $\log p$  stages, where  $p$  is the number of inputs (processing nodes) and also the number of outputs (memory banks). Each stage of the omega network consists of an interconnection pattern that connects  $p$  inputs and  $p$  outputs; a link exists between input  $i$  and output  $j$  if the following is true:

$$\text{Equation 1 : } \quad j=2i, \text{ if } 0 \leq i \leq p/2-1$$

$$\quad \quad \quad j=2i+1-p, \text{ if } p/2 \leq i \leq p-1$$

Figure . The schematic of a typical multistage interconnection network.



Equation 1 represents a left-rotation operation on the binary representation of  $i$  to obtain  $j$ . This interconnection pattern is called a perfect shuffle. Figure shows a perfect shuffle interconnection pattern for eight inputs and outputs. At each stage of an omega network, a perfect shuffle interconnection pattern feeds into a set of  $p/2$  switches or switching nodes. Each switch is in one of two connection modes. In one mode, the inputs are sent straight through to the outputs, as shown in Figure (a). This is called the pass-through connection. In the other mode, the inputs to the switching node are crossed over and then sent out, as shown in the figure (b). This is called the cross-over connection.

Figure . A perfect shuffle interconnection for eight inputs and outputs.

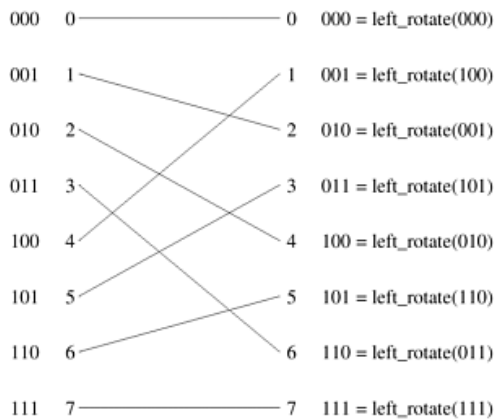
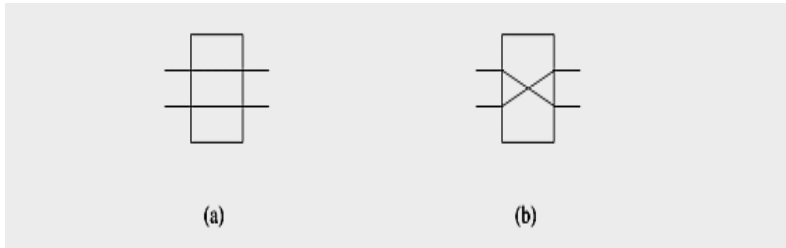


Figure . Two switching configurations of the 2 x 2 switch: (a) Pass-through; (b) Cross-over.



An omega network has  $p/2 \times \log p$  switching nodes, and the cost of such a network grows as  $\Theta(p \log p)$ . Note that this cost is less than the  $\Theta(p^2)$  cost of a complete crossbar network. Figure shows an omega network for eight processors (denoted by the binary numbers on the left) and eight memory banks (denoted by the binary numbers on the right). Routing data in an omega network is accomplished using a simple scheme. Let  $s$  be the binary representation of a processor that needs to write some data into memory bank  $t$ . The data traverses the link to the first switching node. If the most significant bits of  $s$  and  $t$  are the same, then the data is routed in pass-through mode by the switch. If these bits are different, then the data is routed through in crossover mode. This scheme is repeated at the next switching stage using the next most significant bit. Traversing  $\log p$  stages uses all  $\log p$  bits in the binary representations of  $s$  and  $t$ .

Figure . A complete omega network connecting eight inputs and eight outputs.

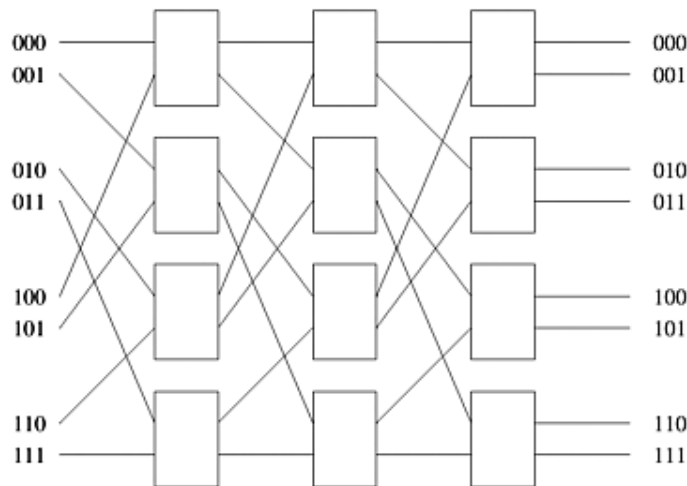


Figure shows data routing over an omega network from processor two (010) to memory bank seven (111) and from processor six (110) to memory bank four (100). This figure also illustrates an important property of this network. When processor two (010) is communicating with memory bank seven (111), it blocks the path from processor six (110) to memory bank four (100). Communication link AB is used by both communication paths. Thus, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as blocking networks.

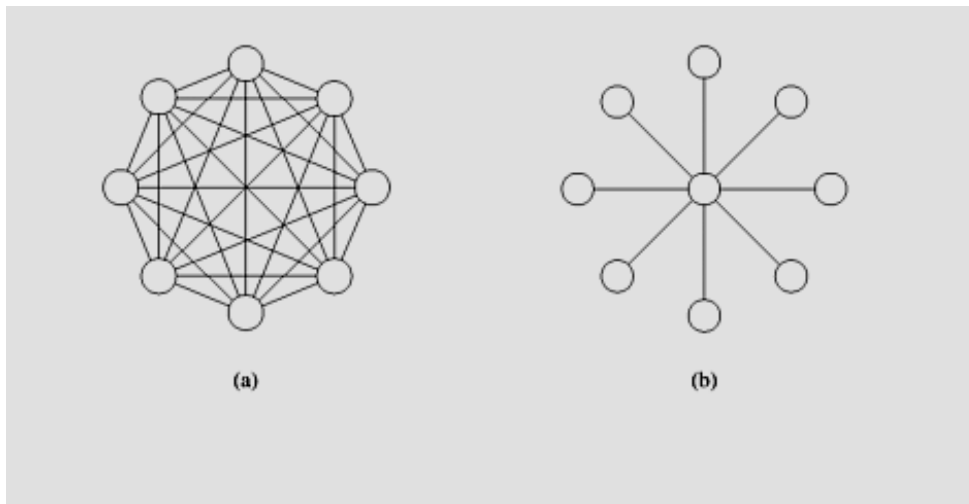


## Lecture ,#11,#12

### Completely-Connected Network

In a completely-connected network, each node has a direct communication link to every other node in the network. Figure (a) illustrates a completely-connected network of eight nodes. This network is ideal in the sense that a node can send a message to another node in a single step, since a communication link exists between them. Completely-connected networks are the static counterparts of crossbar switching networks, since in both networks, the communication between any input/output pair does not block communication between any other pair.

Figure : (a) A completely-connected network of eight nodes; (b) a star connected network of nine nodes.



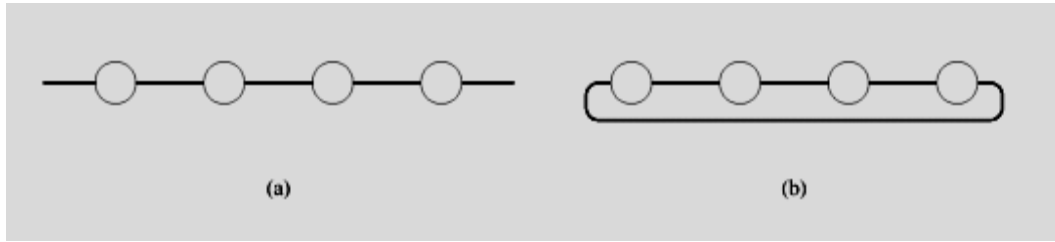
### Star-Connected Network

In a star-connected network, one processor acts as the central processor. Every other processor has a communication link connecting it to this processor. Figure (b) shows a star-connected network of nine processors. The star-connected network is similar to bus-based networks. Communication between any pair of processors is routed through the central processor, just as the shared bus forms the medium for all communication in a bus-based network. The central processor is the bottleneck in the star topology.

### Linear Arrays, Meshes, and k-d Meshes

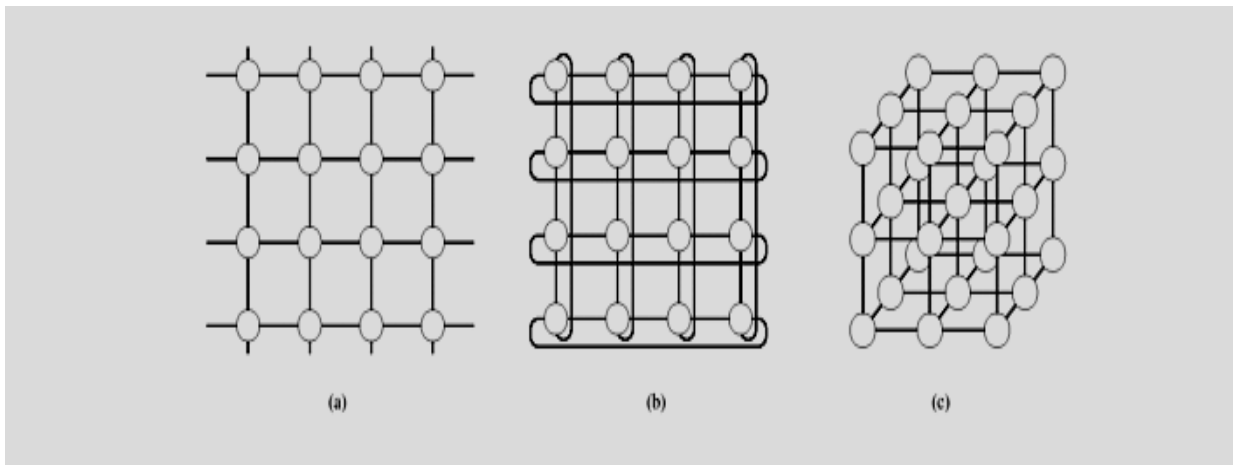
Due to the large number of links in completely connected networks, sparser networks are typically used to build parallel computers. A family of such networks spans the space of linear arrays and hypercubes. A linear array is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. A simple extension of the linear array (Figure (a)) is the ring or a 1-D torus (Figure (b)). The ring has a wraparound connection between the extremities of the linear array. In this case, each node has two neighbors.

Figure : Linear arrays: (a) with no wraparound links; (b) with wraparound link.



A two-dimensional mesh illustrated in the figure is an extension of the linear array to two-dimensions. Each dimension has  $\sqrt{p}$  nodes with a node identified by a two-tuple  $(i, j)$ . Every node (except those on the periphery) is connected to four other nodes whose indices differ in any dimension by one. A 2-D mesh has the property that it can be laid out in 2-D space, making it attractive from a wiring standpoint. Furthermore, a variety of regularly structured computations map very naturally to a 2-D mesh. For this reason, 2-D meshes were often used as interconnects in parallel machines. Two dimensional meshes can be augmented with wraparound links to form two dimensional tori illustrated in the figure. The three-dimensional cube is a generalization of the 2-D mesh to three dimensions, as illustrated in Figure (c). Each node element in a 3-D cube, with the exception of those on the periphery, is connected to six other nodes, two along each of the three dimensions. A variety of physical simulations commonly executed on parallel computers (for example, 3-D weather modeling, structural modeling, etc.) can be mapped naturally to 3-D network topologies. For this reason, 3-D cubes are used commonly in interconnection networks for parallel computers (for example, in the Cray T3E).

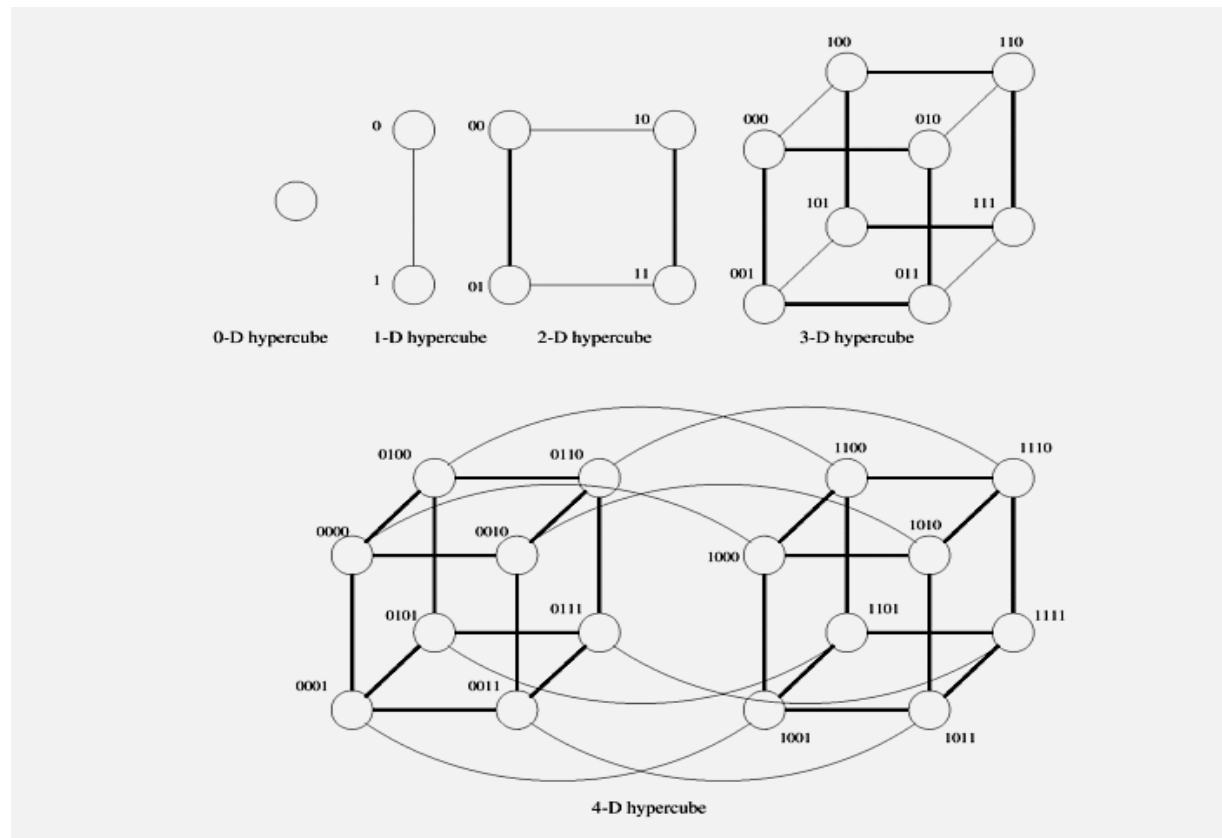
**Figure :** Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.



The general class of  $k$ - $d$  meshes refers to the class of topologies consisting of  $d$  dimensions with  $k$  nodes along each dimension. Just as a linear array forms one extreme of the  $k$ - $d$  mesh family, the other extreme is formed by an interesting topology called the hypercube. The hypercube topology has two nodes along each dimension and  $\log p$  dimensions. The construction of a hypercube is illustrated in the figure . A zero-dimensional hypercube consists of  $2^0$ , i.e., one node. A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them. A two-dimensional hypercube of four nodes is constructed from two one-dimensional hypercubes by connecting corresponding nodes. In general a  $d$ -dimensional hypercube is constructed by connecting

corresponding nodes of two  $(d - 1)$  dimensional hypercubes. The following figure illustrates this for up to 16 nodes in a 4-D hypercube.

Figure . Construction of hypercubes from hypercubes of lower dimension.

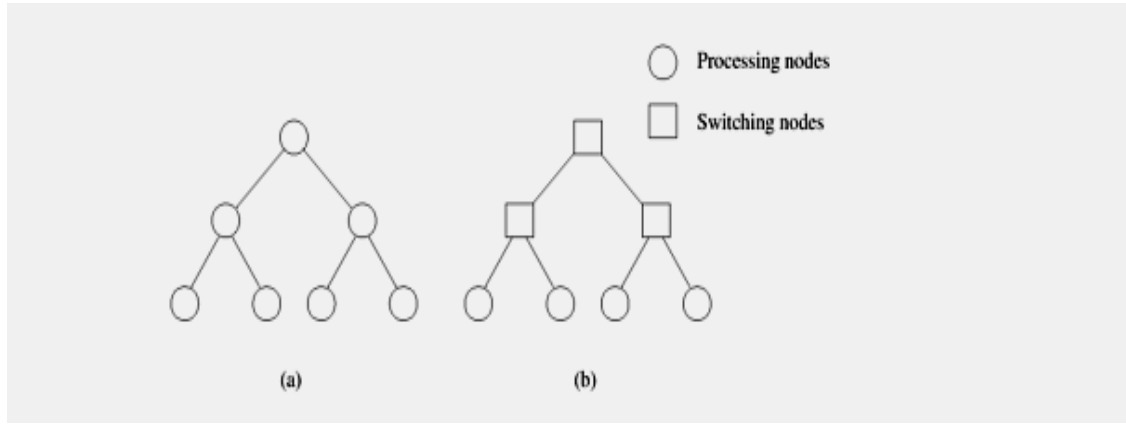


It is useful to derive a numbering scheme for nodes in a hypercube. A simple numbering scheme can be derived from the construction of a hypercube. As illustrated in the figure, if we have a numbering of two subcubes of  $p/2$  nodes, we can derive a numbering scheme for the cube of  $p$  nodes by prefixing the labels of one of the subcubes with a "0" and the labels of the other subcube with a "1". This numbering scheme has the useful property that the minimum distance between two nodes is given by the number of bits that are different in the two labels. For example, nodes labeled 0110 and 0101 are two links apart, since they differ at two bit positions. This property is useful for deriving a number of parallel algorithms for the hypercube architecture.

### Tree-Based Networks

A tree network is one in which there is only one path between any pair of nodes. Both linear arrays and star-connected networks are special cases of tree networks. Figure shows networks based on complete binary trees. Static tree networks have a processing element at each node of the tree (Figure (a)). Tree networks also have a dynamic counterpart. In a dynamic tree network, nodes at intermediate levels are switching nodes and the leaf nodes are processing elements (Figure (b)).

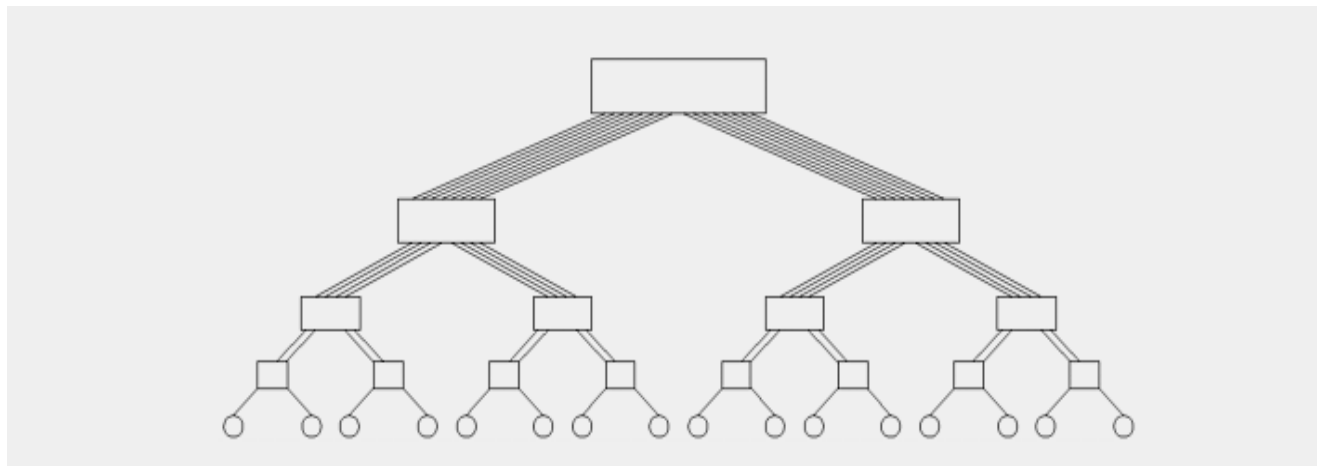
Figure 2.18. Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.



To route a message in a tree, the source node sends the message up the tree until it reaches the node at the root of the smallest subtree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node.

Tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many nodes in the left subtree of a node communicate with nodes in the right subtree, the root node must handle all the messages. This problem can be alleviated in dynamic tree networks by increasing the number of communication links and switching nodes closer to the root. This network, also called a fat tree, is illustrated in the following figure.

**Figure 2.19. A fat tree network of 16 processing nodes.**





## Lecture #13,#14

### Evaluating Static Interconnection Networks

We now discuss various criteria used to characterize the cost and performance of static interconnection networks. We use these criteria to evaluate static networks introduced in the previous subsection.

**Diameter:** The diameter of a network is the maximum distance between any two processing nodes in the network. The distance between two processing nodes is defined as the shortest path (in terms of number of links) between them. The diameter of a completely-connected network is one, and that of a star-connected network is two. The diameter of a ring network is  $\lfloor p/2 \rfloor$ . The diameter of a two-dimensional mesh without wraparound connections is  $2(\sqrt{p}-1)$  for the two nodes at diagonally opposed corners, and that of a wraparound mesh is  $2\lfloor \sqrt{p}/2 \rfloor$ . The diameter of a hypercube-connected network is  $\log p$  since two node labels can differ in at most  $\log p$  positions. The diameter of a complete binary tree is  $2 \log((p+1)/2)$  because the two communicating nodes may be in separate subtrees of the root node, and a message might have to travel all the way to the root and then down the other subtree.

**Connectivity:** The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable, because it lowers contention for communication resources. One measure of connectivity is the minimum number of arcs that must be removed from the network to break it into two disconnected networks. This is called the arc connectivity of the network. The arc connectivity is one for linear arrays, as well as tree and star networks. It is two for rings and 2-D meshes without wraparound, four for 2-D wraparound meshes, and  $d$  for  $d$ -dimensional hypercubes.

**Bisection Width and Bisection Bandwidth:** The bisection width of a network is defined as the minimum number of communication links that must be removed to partition the network into two equal halves. The bisection width of a ring is two, since any partition cuts across only two communication links. Similarly, the bisection width of a two-dimensional  $p$ -node mesh without wraparound connections is  $\sqrt{p}$  and with wraparound connections is  $2\sqrt{p}$ . The bisection width of a tree and a star is one, and that of a completely-connected network of  $p$  nodes is  $p^2/4$ . The bisection width of a hypercube can be derived from its construction. We construct a  $d$ -dimensional hypercube by connecting corresponding links of two  $(d-1)$ -dimensional hypercubes. Since each of these sub-cubes contains  $2^{(d-1)}$  or  $p/2$  nodes, at least  $p/2$  communication links must cross any partition of a hypercube into two sub-cubes.

The number of bits that can be communicated simultaneously over a link connecting two nodes is called the channel width. Channel width is equal to the number of physical wires in each communication link. The peak rate at which a single physical wire can deliver bits is called the channel rate. The peak rate at which data can be communicated between the ends of a communication link is called channel bandwidth. Channel bandwidth is the product of channel rate and channel width.

**Table 1. A summary of the characteristics of various static network topologies connecting p nodes.**

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of link)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	$\sqrt{p}$	2	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lfloor \sqrt{p} / 2 \rfloor$	$2\sqrt{p}$	4	2p
Hypercube	$\log p$	$p/2$	Logp	$(p \log p)/2$
Wraparound k-ary d-cube	$d\lfloor k / 2 \rfloor$	$2k^{d-1}$	2d	dp

The **bisection bandwidth** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as cross-section bandwidth.

**Cost:** Many criteria can be used to evaluate the cost of a network. One way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network. Linear arrays and trees use only  $p - 1$  links to connect  $p$  nodes. A  $d$ -dimensional wraparound mesh has  $dp$  links. A hypercube-connected network has  $(p \log p)/2$  links.

The bisection bandwidth of a network can also be used as a measure of its cost, as it provides a lower bound on the area in a two-dimensional packaging or the volume in a three-dimensional packaging. If the bisection width of a network is  $w$ , the lower bound on the area in a two-dimensional packaging is  $\Theta(w^2)$ , and the lower bound on the volume in a three-dimensional packaging is  $\Theta(w^{3/2})$ . According to this criterion, hypercubes and completely connected networks are more expensive than the other networks.

We summarize the characteristics of various static networks in the Table , which highlights the various cost-performance tradeoffs.

### Evaluating Dynamic Interconnection Networks

A number of evaluation metrics for dynamic networks follow from the corresponding metrics for static networks. Since a message traversing a switch must pay an overhead, it is logical to think of each switch as a node in the network, in addition to the processing nodes. The diameter of the network can now be defined as the maximum distance between any two nodes in the network. This is indicative of the maximum delay that a message will encounter in being communicated between the selected pair of nodes. In reality, we would like the metric to be the maximum distance between any two processing nodes; however, for all networks of interest, this is equivalent to the maximum distance between any (processing or switching) pair of nodes.

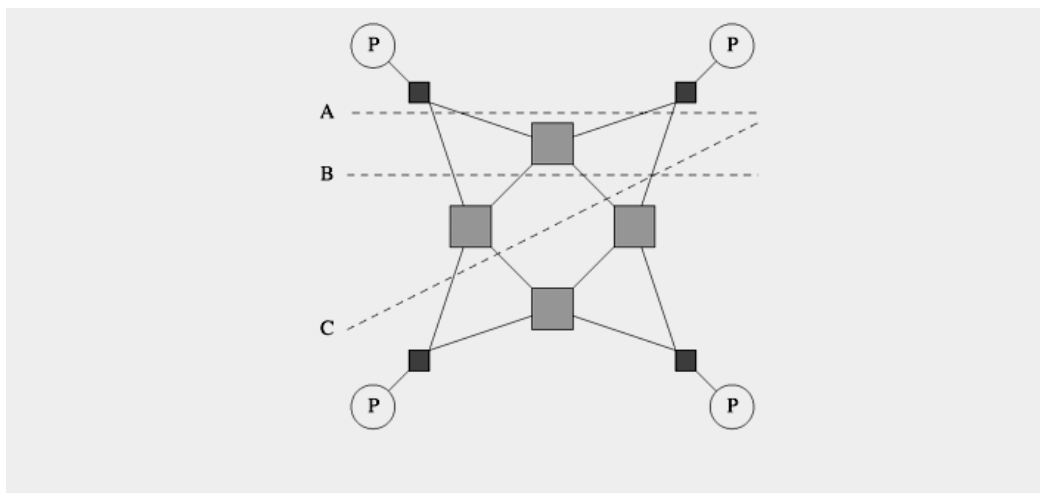
The connectivity of a dynamic network can be defined in terms of node or edge connectivity. The node connectivity is the minimum number of nodes that must fail (be removed from the network) to fragment the network into two parts. As before, we should consider only switching nodes (as opposed to all nodes). However, considering all nodes gives a good approximation to the multiplicity of paths in a dynamic network. The arc connectivity of the network can be similarly defined as the minimum number of edges that must fail (be removed from the network) to fragment the network into two unreachable parts.

The bisection width of a dynamic network must be defined more precisely than diameter and connectivity. In the case of bisection width, we consider any possible partitioning of the  $p$  processing nodes into two equal parts. Note that this does not restrict the partitioning of the switching nodes. For each such partition, we select an induced partitioning of the switching nodes such that the number of edges crossing this partition is minimized. The minimum number of edges for any such partition is the bisection width of the dynamic network. Another intuitive way of thinking of bisection width is in terms of the minimum number of edges that must be removed from the network so as to partition the network into two halves with identical number of processing nodes. We illustrate this concept further in the following example:

Example : Bisection width of dynamic networks

Consider the network illustrated in the figure. We illustrate here three bisections, A, B, and C, each of which partitions the network into two groups of two processing nodes each. Notice that these partitions need not partition the network nodes equally. In the example, each partition results in an edge cut of four. We conclude that the bisection width of this graph is four.

Figure . Bisection width of a dynamic network is computed by examining various equi-partitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four. Therefore, the bisection width of this graph is four.



The cost of a dynamic network is determined by the link cost, as is the case with static networks, as well as the switch cost. In typical dynamic networks, the degree of a switch is constant. Therefore, the

number of links and switches is asymptotically identical. Furthermore, in typical networks, switch cost exceeds link cost. For this reason, the cost of dynamic networks is often determined by the number of switching nodes in the network.

We summarize the characteristics of various dynamic networks in the table.

**Table : A summary of the characteristics of various dynamic network topologies connecting  $p$  processing nodes.**

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	$P$	1	$p^2$
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

### Lecture #15,#16,#17

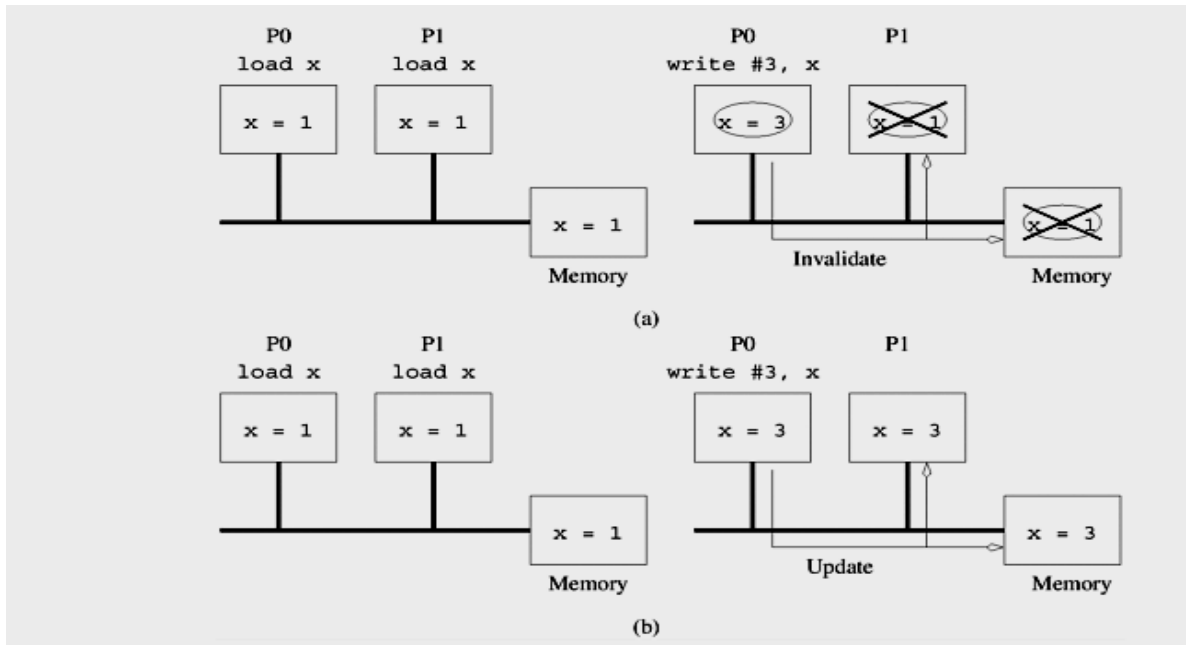
#### Cache Coherence in Multiprocessor Systems

While interconnection networks provide basic mechanisms for communicating messages (data), in the case of shared-address-space computers additional hardware is required to keep multiple copies of data consistent with each other. Specifically, if there exist two copies of the data (in different caches/memory elements), how do we ensure that different processors operate on these in a manner that follows predefined semantics?

The problem of keeping caches in multiprocessor systems coherent is significantly more complex than in uniprocessor systems. This is because in addition to multiple copies as in uniprocessor systems, there may also be multiple processors modifying these copies. Consider a simple scenario illustrated in the figure. Two processors  $P_0$  and  $P_1$  are connected over a shared bus to a globally accessible memory. Both processors load the same variable. There are now three copies of the variable. The coherence mechanism must now ensure that all operations performed on these copies are serializable (i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule). When a processor changes the value of its copy of the variable, one of two things must happen: the other copies must be invalidated, or the other copies must be updated. Failing this, other processors may potentially work with incorrect (stale) values of the variable. These two protocols are referred to as invalidate and update protocols and are illustrated in Figure (a) and (b).

**Figure . Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.**





In an update protocol, whenever a data item is written, all of its copies in the system are updated. For this reason, if a processor simply reads a data item once and never uses it, subsequent updates to this item at other processors cause excess overhead in terms of latency at source and bandwidth on the network. On the other hand, in this situation, an invalidate protocol invalidates the data item on the first update at a remote processor and subsequent updates need not be performed on this copy.

Another important factor affecting the performance of these protocols is false sharing. False sharing refers to the situation in which different processors update different parts of the same cache-line. Thus, although the updates are not performed on shared variables, the system does not detect this. In an invalidate protocol, when a processor updates its part of the cache-line, the other copies of this line are invalidated. When other processors try to update their parts of the cache-line, the line must actually be fetched from the remote processor. It is easy to see that false-sharing can cause a cache-line to be ping-ponged between various processors. In an update protocol, this situation is slightly better since all reads can be performed locally and the writes must be updated. This saves an invalidate operation that is otherwise wasted.

The tradeoff between invalidate and update schemes is the classic tradeoff between communication overhead (updates) and idling (stalling in invalidates). Current generation cache coherent machines typically rely on invalidate protocols. The rest of our discussion of multiprocessor cache systems therefore assumes invalidate protocols.

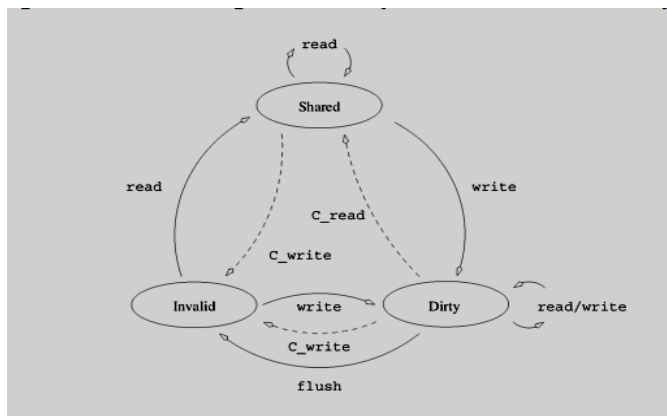
**Maintaining Coherence Using Invalidate Protocols** Multiple copies of a single data item are kept consistent by keeping track of the number of copies and the state of each of these copies. We discuss here one possible set of states associated with data items and events that trigger transitions among these states. Note that this set of states and transitions is not unique. It is possible to define other states and associated transitions as well.

Let us revisit the example in the figure. Initially the variable  $x$  resides in the global memory. The first step executed by both processors is a load operation on this variable. At this point, the state

of the variable is said to be shared, since it is shared by multiple processors. When processor  $P_0$  executes a store on this variable, it marks all other copies of this variable as invalid. It must also mark its own copy as modified or dirty. This is done to ensure that all subsequent accesses to this variable at other processors will be serviced by processor  $P_0$  and not from the memory. At this point, say, processor  $P_1$  executes another load operation on  $x$ . Processor  $P_1$  attempts to fetch this variable and, since the variable was marked dirty by processor  $P_0$ , processor  $P_0$  services the request. Copies of this variable at processor  $P_1$  and the global memory are updated and the variable re-enters the shared state. Thus, in this simple model, there are three states - shared, invalid, and dirty - that a cache line goes through.

The complete state diagram of a simple three-state protocol is illustrated in the figure. The solid lines depict processor actions and the dashed lines coherence actions. For example, when a processor executes a read on an invalid block, the block is fetched and a transition is made from invalid to shared. Similarly, if a processor does a write on a shared block, the coherence protocol propagates a  $C\_write$  (a coherence write) on the block. This triggers a transition from shared to invalid at all the other blocks.

**Figure State diagram of a simple three-state coherence protocol.**



Example : Maintaining coherence using a simple three-state protocol

Consider an example of two program segments being executed by processor  $P_0$  and  $P_1$  as illustrated in the figure. The system consists of local memories (or caches) at processors  $P_0$  and  $P_1$ , and a global memory. The three-state protocol assumed in this example corresponds to the state diagram illustrated in the figure. Cache lines in this system can be either shared, invalid, or dirty. Each data item (variable) is assumed to be on a different cache line. Initially, the two variables  $x$  and  $y$  are tagged dirty and the only copies of these variables exist in the global memory. The figure illustrates state transitions along with values of copies of the variables with each instruction execution.

**Figure . Example of parallel program execution with the simple three-state coherence protocol**

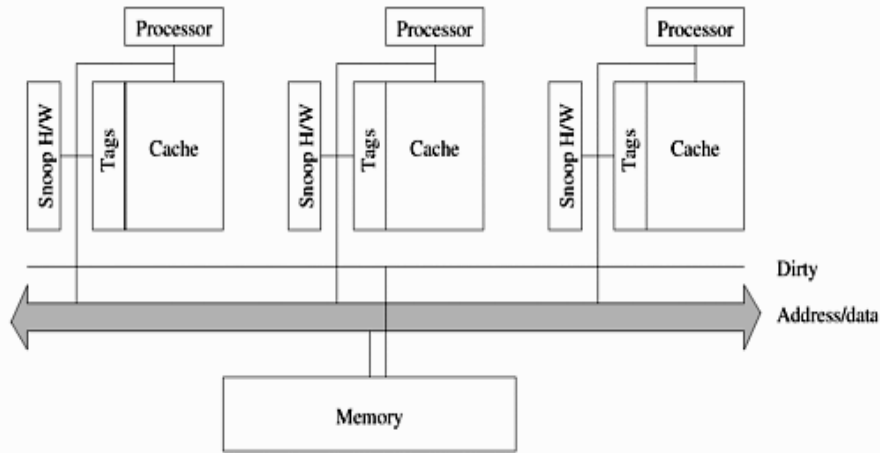
Time ↓	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D y = 12, D
	read x	read y	x = 5, S	y = 12, S	x = 5, S y = 12, S
	x = x + 1	y = y + 1	x = 6, D	y = 13, D	x = 5, I y = 12, I
	read y	read x	y = 13, S x = 6, S	y = 13, S x = 6, S	y = 13, S x = 6, S
	x = x + y	y = x + y	x = 19, D y = 13, I	x = 6, I y = 19, D	x = 6, I y = 13, I
	x = x + 1	y = y + 1	x = 20, D	y = 20, D	x = 6, I y = 13, I

The implementation of coherence protocols can be carried out using a variety of hardware mechanisms – snoopy systems, directory based systems, or combinations thereof.

### Snoopy Cache Systems

Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. In such systems, all processors snoop on (monitor) the bus for transactions. This allows the processor to make state transitions for its cache-blocks. The figure illustrates a typical snoopy bus based system. Each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks. These tags are updated according to the state diagram associated with the coherence protocol. For instance, when the snoop hardware detects that a read has been issued to a cache block that it has a dirty copy of, it asserts control of the bus and puts the data out. Similarly, when the snoop hardware detects that a write operation has been issued on a cache block that it has a copy of, it invalidates the block. Other state transitions are made in this fashion locally.

Figure 2.24. A simple snoopy bus based cache coherence system.



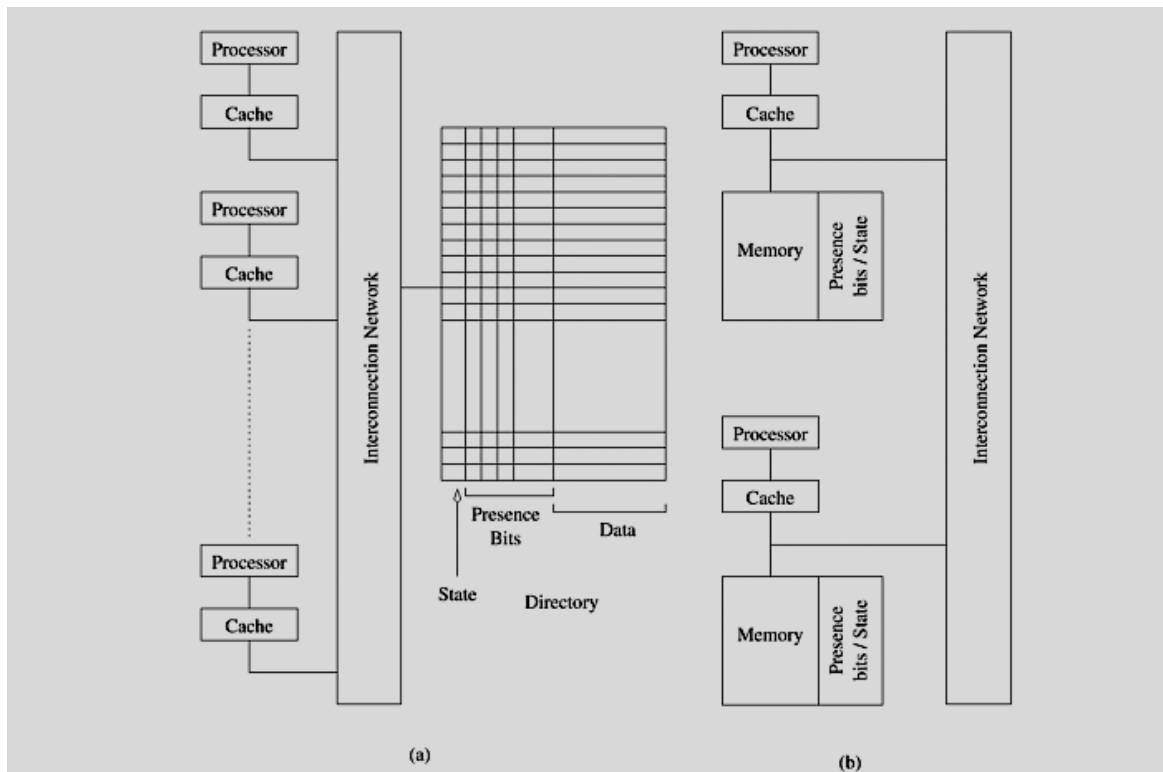
Performance of Snoopy Caches Snoopy protocols have been extensively studied and used in commercial systems. This is largely because of their simplicity and the fact that existing bus based systems can be upgraded to accommodate snoopy protocols. The performance gains of snoopy systems are derived from the fact that if different processors operate on different data items, these items can be cached. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic. Similarly, if a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local. In both cases, the coherence protocol does not add any overhead. On the other hand, if multiple processors read and update the same data item, they generate coherence functions across processors. Since a shared bus has a finite bandwidth, only a constant number of such coherence operations can execute in unit time. This presents a fundamental bottleneck for snoopy bus based systems.

Snoopy protocols are intimately tied to multicomputers based on broadcast networks such as buses. This is because all processors must snoop all the messages. Clearly, broadcasting all of a processor's memory operations to all the processors is not a scalable solution. An obvious solution to this problem is to propagate coherence operations only to those processors that must participate in the operation (i.e., processors that have relevant copies of the data). This solution requires us to keep track of which processors have copies of various data items and also the relevant state information for these data items. This information is stored in a directory, and the coherence mechanism based on such information is called a directory-based system.

### Directory Based Systems

Consider a simple system in which the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached. These bitmap entries are sometimes referred to as the presence bits. As before, we assume a three-state protocol with the states labeled invalid, dirty, and shared. The key to the performance of directory based schemes is the simple observation that only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory.


Figure . Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.



Revisiting the code segment in the figure, when processors  $P_0$  and  $P_1$  access the block corresponding to variable  $x$ , the state of the block is changed to shared, and the presence bits updated to indicate that processors  $P_0$  and  $P_1$  share the block. When  $P_0$  executes a store on the variable, the state in the directory is changed to dirty and the presence bit of  $P_1$  is reset. All subsequent operations on this variable performed at processor  $P_0$  can proceed locally. If another processor reads the value, the directory notices the dirty tag and uses the presence bits to direct the request to the appropriate processor. Processor  $P_0$  updates the block in the memory, and sends it to the requesting processor. The presence bits are modified to reflect this and the state transitions to shared.

Performance of Directory Based Schemes as is the case with snoopy protocols, if different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally. Furthermore, if multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Coherence actions are initiated when multiple processors attempt to update the same data item. In this case, in addition to the necessary data movement, coherence operations add to the overhead in the form of propagation of state updates (invalidates or updates) and generation of state information from the directory. The former takes the form of communication overhead and the latter adds contention. The communication overhead is a function of the number of processors requiring state updates and the algorithm for propagating state information. The contention overhead is more fundamental in nature. Since the directory is in memory and the memory system can only service a bounded number of read/write operations in unit time, the number of state updates is ultimately



bounded by the directory. If a parallel program requires a large number of coherence actions (large number of read/write shared data blocks) the directory will ultimately bound its parallel performance.

Finally, from the point of view of cost, the amount of memory required to store the directory may itself become a bottleneck as the number of processors increases. Recall that the directory size grows as  $O(mp)$ , where  $m$  is the number of memory blocks and  $p$  the number of processors. One solution would be to make the memory block larger (thus reducing  $m$  for a given memory size). However, this adds to other overheads such as false sharing, where two processors update distinct data items in a program but the data items happen to lie in the same memory block.

Since the directory forms a central point of contention, it is natural to break up the task of maintaining coherence across multiple processors. The basic principle is to let each processor maintain coherence of its own memory blocks, assuming a physical (or logical) partitioning of the memory blocks across processors. This is the principle of a distributed directory system.

**Distributed Directory Schemes** In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. The architecture of such a system is illustrated in Figure (b). Since each memory block has an owner (which can typically be computed from the block address), its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. Similarly, when a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. In this way, the directory is decentralized and the contention associated with the central directory is alleviated. Note that the communication overhead associated with state update messages is not reduced.

**Performance of Distributed Directory Schemes** As is evident, distributed directories permit  $O(p)$  simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.



## Lecture # 18

### Communication Costs in Parallel Machines

One of the major overheads in the execution of parallel programs arises from communication of information between processing elements. The cost of communication is dependent on a variety of features including the programming model semantics, the network topology, data handling and routing, and associated software protocols. These issues form the focus of our discussion here.

#### Message Passing Costs in Parallel Computers

The time taken to communicate a message between two nodes in a network is the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

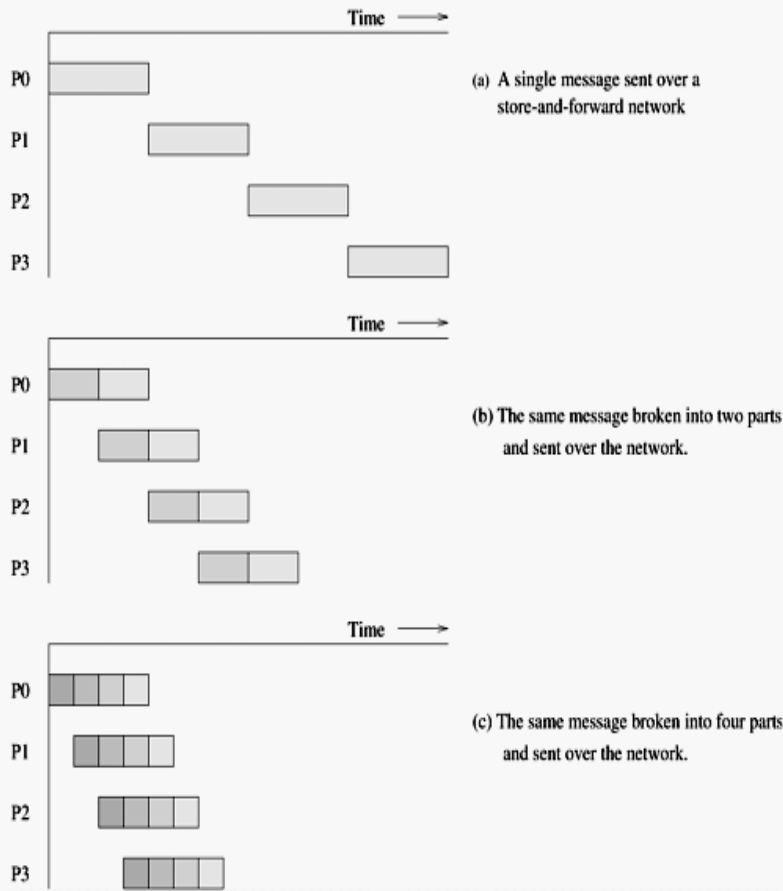
1. **Startup time ( $t_s$ ):** The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local node and the router. This delay is incurred only once for a single message transfer.
2. **Per-hop time ( $t_h$ ):** After a message leaves a node, it takes a finite amount of time to reach the next node in its path. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. It is also known as node latency. The per-hop time is directly related to the latency within the routing switch for determining which output buffer or channel the message should be forwarded to.
3. **Per-word transfer time ( $t_w$ ):** If the channel bandwidth is  $r$  words per second, then each word takes time  $t_w = 1/r$  to traverse the link. This time is called the per-word transfer time. This time includes network as well as buffering overheads.

We now discuss two routing techniques that have been used in parallel computers – store-and-forward routing and cut-through routing.

#### Store-and-Forward Routing

In store-and-forward routing, when a message is traversing a path with multiple links, each intermediate node on the path forwards the message to the next node after it has received and stored the entire message. Figure (a) shows the communication of a message through a store-and-forward network.

Figure . Passing a message from node  $P_0$  to  $P_3$  (a) through a store-and-forward communication network; (b) and (c) extending the concept to cut-through routing. The shaded regions represent the time that the message is in transit. The startup time associated with this message transfer is assumed to be zero.



Suppose that a message of size  $m$  is being transmitted through such a network. Assume that it traverses  $l$  links. At each link, the message incurs a cost  $t_h$  for the header and  $t_w m$  for the rest of the message to traverse the link. Since there are  $l$  such links, the total time is  $(t_h + t_w m)l$ . Therefore, for store-and-forward routing, the total communication cost for a message of size  $m$  words to traverse  $l$  communication links is

$$t_{\text{comm}} = t_s + (t_h + t_w m)l \quad (1)$$

In current parallel computers, the per-hop time  $t_h$  is quite small. For most parallel algorithms, it is less than  $t_w m$  even for small values of  $m$  and thus can be ignored. For parallel platforms using store-and-forward routing, the time given by (1) can be simplified to

$$t_{\text{comm}} = t_s + t_w m l$$

### Packet Routing

Store-and-forward routing makes poor use of communication resources. A message is sent from one node to the next only after the entire message has been received (Figure (a)). Consider the scenario shown in Figure (b), in which the original message is broken into two equal sized parts before it is sent. In this case, an intermediate node waits for only half of the original message to arrive before passing it on. The increased utilization of communication resources and reduced communication time



is apparent from Figure (b). Figure (c) goes a step further and breaks the message into four parts. In addition to better utilization of communication resources, this principle offers other advantages – lower overhead from packet loss (errors), possibility of packets taking different paths, and better error correction capability. For these reasons, this technique is the basis for long-haul communication networks such as the Internet, where error rates, number of hops, and variation in network state can be higher. Of course, the overhead here is that each packet must carry routing, error correction, and sequencing information.

Consider the transfer of an  $m$  word message through the network. The time taken for programming the network interfaces and computing the routing information, etc., is independent of the message length. This is aggregated into the startup time  $t_s$  of the message transfer. We assume a scenario in which routing tables are static over the time of message transfer (i.e., all packets traverse the same path). While this is not a valid assumption under all circumstances, it serves the purpose of motivating a cost model for message transfer. The message is broken into packets, and packets are assembled with their error, routing, and sequencing fields. The size of a packet is now given by  $r + s$ , where  $r$  is the original message and  $s$  is the additional information carried in the packet. The time for packetizing the message is proportional to the length of the message. We denote this time by  $mt_{w1}$ . If the network is capable of communicating one word every  $t_{w2}$  seconds, incurs a delay of  $t_h$  on each hop, and if the first packet traverses  $l$  hops, then this packet takes time  $t_h l + t_{w2}(r + s)$  to reach the destination. After this time, the destination node receives an additional packet every  $t_{w2}(r + s)$  seconds. Since there are  $m/r - 1$  additional packets, the total communication time is given by:

$$\begin{aligned} t_{\text{comm}} &= t_s + mt_{w1} + t_h l + t_{w2}(r + s) + (m/r - 1) t_{w2}(r + s) \\ &= t_s + mt_{w1} + t_h l + t_{w2}m + (s/r)m t_{w2} \\ &= t_s + t_h l + m t_w \end{aligned}$$

Where

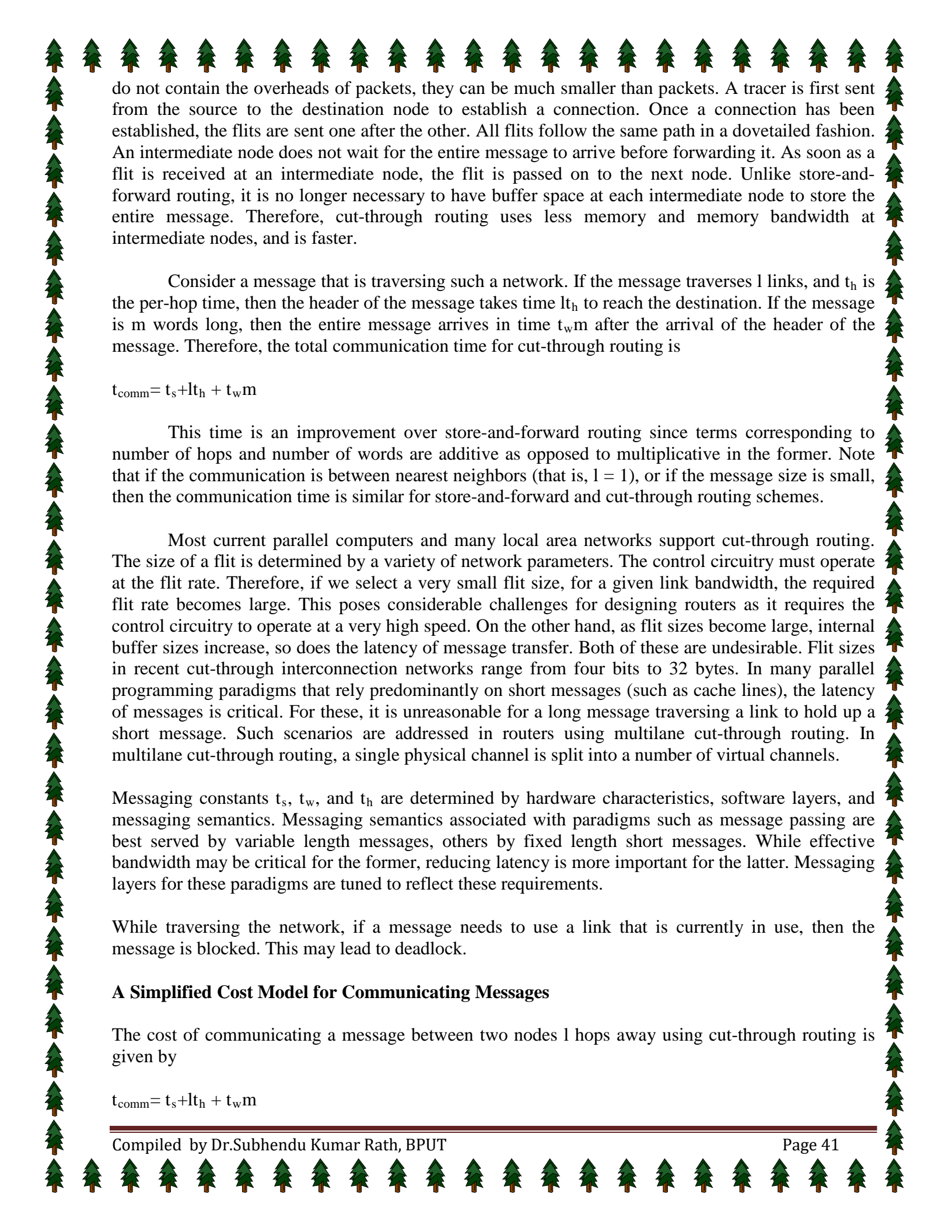
$$t_w = mt_{w1} + t_{w2}m + (s/r)m t_{w2}$$

Packet routing is suited to networks with highly dynamic states and higher error rates, such as local- and wide-area networks. This is because individual packets may take different routes and retransmissions can be localized to lost packets.

### Cut-Through Routing

In interconnection networks for parallel computers, additional restrictions can be imposed on message transfers to further reduce the overheads associated with packet switching. By forcing all packets to take the same path, we can eliminate the overhead of transmitting routing information with each packet. By forcing in-sequence delivery, sequencing information can be eliminated. By associating error information at message level rather than packet level, the overhead associated with error detection and correction can be reduced. Finally, since error rates in interconnection networks for parallel machines are extremely low, lean error detection mechanisms can be used instead of expensive error correction schemes.

The routing scheme resulting from these optimizations is called cut-through routing. In cut-through routing, a message is broken into fixed size units called flow control digits or flits. Since flits



do not contain the overheads of packets, they can be much smaller than packets. A tracer is first sent from the source to the destination node to establish a connection. Once a connection has been established, the flits are sent one after the other. All flits follow the same path in a dovetailed fashion. An intermediate node does not wait for the entire message to arrive before forwarding it. As soon as a flit is received at an intermediate node, the flit is passed on to the next node. Unlike store-and-forward routing, it is no longer necessary to have buffer space at each intermediate node to store the entire message. Therefore, cut-through routing uses less memory and memory bandwidth at intermediate nodes, and is faster.

Consider a message that is traversing such a network. If the message traverses  $l$  links, and  $t_h$  is the per-hop time, then the header of the message takes time  $lt_h$  to reach the destination. If the message is  $m$  words long, then the entire message arrives in time  $t_w m$  after the arrival of the header of the message. Therefore, the total communication time for cut-through routing is

$$t_{\text{comm}} = t_s + lt_h + t_w m$$

This time is an improvement over store-and-forward routing since terms corresponding to number of hops and number of words are additive as opposed to multiplicative in the former. Note that if the communication is between nearest neighbors (that is,  $l = 1$ ), or if the message size is small, then the communication time is similar for store-and-forward and cut-through routing schemes.

Most current parallel computers and many local area networks support cut-through routing. The size of a flit is determined by a variety of network parameters. The control circuitry must operate at the flit rate. Therefore, if we select a very small flit size, for a given link bandwidth, the required flit rate becomes large. This poses considerable challenges for designing routers as it requires the control circuitry to operate at a very high speed. On the other hand, as flit sizes become large, internal buffer sizes increase, so does the latency of message transfer. Both of these are undesirable. Flit sizes in recent cut-through interconnection networks range from four bits to 32 bytes. In many parallel programming paradigms that rely predominantly on short messages (such as cache lines), the latency of messages is critical. For these, it is unreasonable for a long message traversing a link to hold up a short message. Such scenarios are addressed in routers using multilane cut-through routing. In multilane cut-through routing, a single physical channel is split into a number of virtual channels.

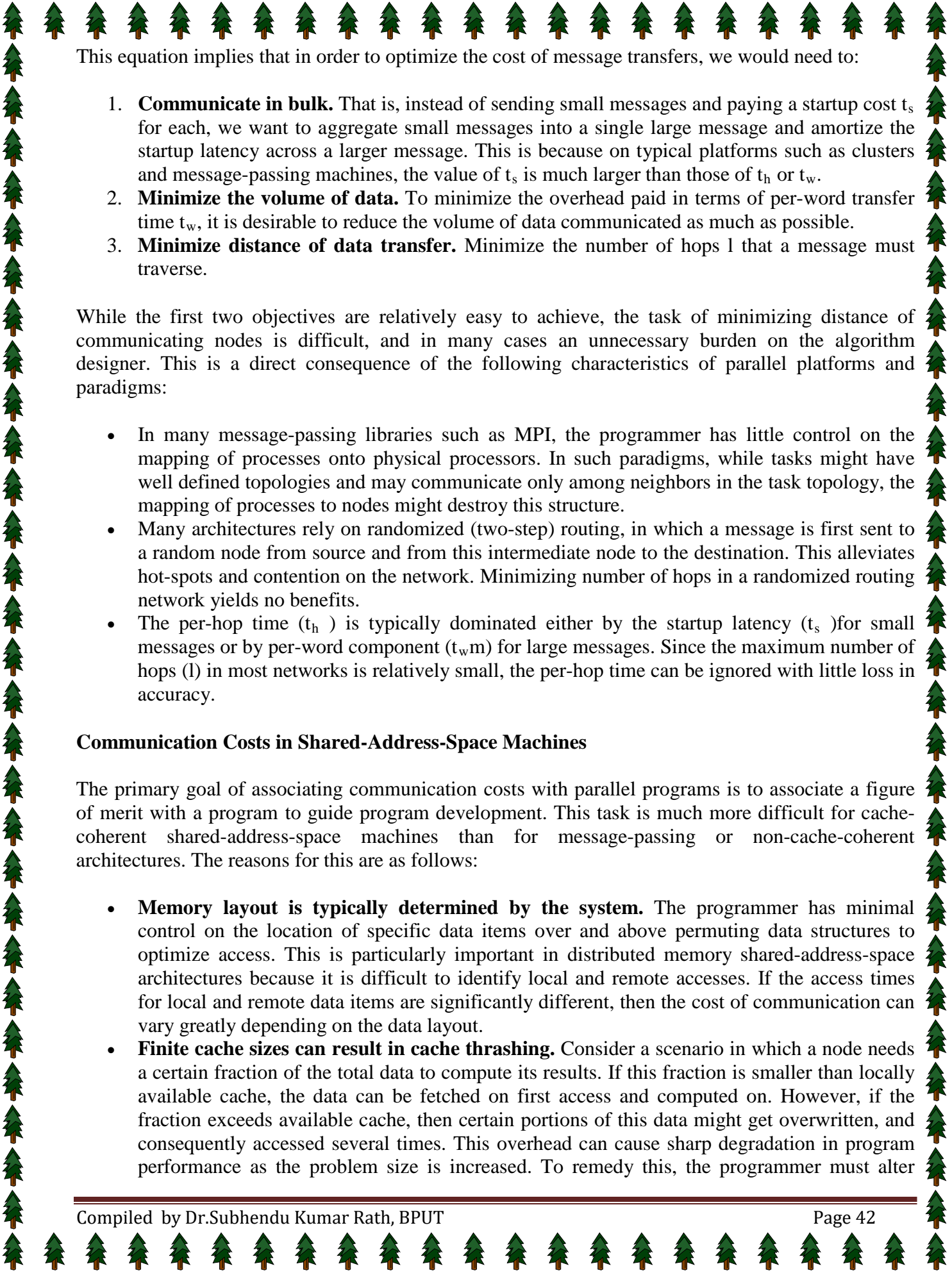
Messaging constants  $t_s$ ,  $t_w$ , and  $t_h$  are determined by hardware characteristics, software layers, and messaging semantics. Messaging semantics associated with paradigms such as message passing are best served by variable length messages, others by fixed length short messages. While effective bandwidth may be critical for the former, reducing latency is more important for the latter. Messaging layers for these paradigms are tuned to reflect these requirements.

While traversing the network, if a message needs to use a link that is currently in use, then the message is blocked. This may lead to deadlock.

### A Simplified Cost Model for Communicating Messages

The cost of communicating a message between two nodes  $l$  hops away using cut-through routing is given by

$$t_{\text{comm}} = t_s + lt_h + t_w m$$



This equation implies that in order to optimize the cost of message transfers, we would need to:

1. **Communicate in bulk.** That is, instead of sending small messages and paying a startup cost  $t_s$  for each, we want to aggregate small messages into a single large message and amortize the startup latency across a larger message. This is because on typical platforms such as clusters and message-passing machines, the value of  $t_s$  is much larger than those of  $t_h$  or  $t_w$ .
2. **Minimize the volume of data.** To minimize the overhead paid in terms of per-word transfer time  $t_w$ , it is desirable to reduce the volume of data communicated as much as possible.
3. **Minimize distance of data transfer.** Minimize the number of hops  $l$  that a message must traverse.


While the first two objectives are relatively easy to achieve, the task of minimizing distance of communicating nodes is difficult, and in many cases an unnecessary burden on the algorithm designer. This is a direct consequence of the following characteristics of parallel platforms and paradigms:

- In many message-passing libraries such as MPI, the programmer has little control on the mapping of processes onto physical processors. In such paradigms, while tasks might have well defined topologies and may communicate only among neighbors in the task topology, the mapping of processes to nodes might destroy this structure.
- Many architectures rely on randomized (two-step) routing, in which a message is first sent to a random node from source and from this intermediate node to the destination. This alleviates hot-spots and contention on the network. Minimizing number of hops in a randomized routing network yields no benefits.
- The per-hop time ( $t_h$ ) is typically dominated either by the startup latency ( $t_s$ ) for small messages or by per-word component ( $t_w m$ ) for large messages. Since the maximum number of hops ( $l$ ) in most networks is relatively small, the per-hop time can be ignored with little loss in accuracy.

### Communication Costs in Shared-Address-Space Machines

The primary goal of associating communication costs with parallel programs is to associate a figure of merit with a program to guide program development. This task is much more difficult for cache-coherent shared-address-space machines than for message-passing or non-cache-coherent architectures. The reasons for this are as follows:

- **Memory layout is typically determined by the system.** The programmer has minimal control on the location of specific data items over and above permuting data structures to optimize access. This is particularly important in distributed memory shared-address-space architectures because it is difficult to identify local and remote accesses. If the access times for local and remote data items are significantly different, then the cost of communication can vary greatly depending on the data layout.
- **Finite cache sizes can result in cache thrashing.** Consider a scenario in which a node needs a certain fraction of the total data to compute its results. If this fraction is smaller than locally available cache, the data can be fetched on first access and computed on. However, if the fraction exceeds available cache, then certain portions of this data might get overwritten, and consequently accessed several times. This overhead can cause sharp degradation in program performance as the problem size is increased. To remedy this, the programmer must alter

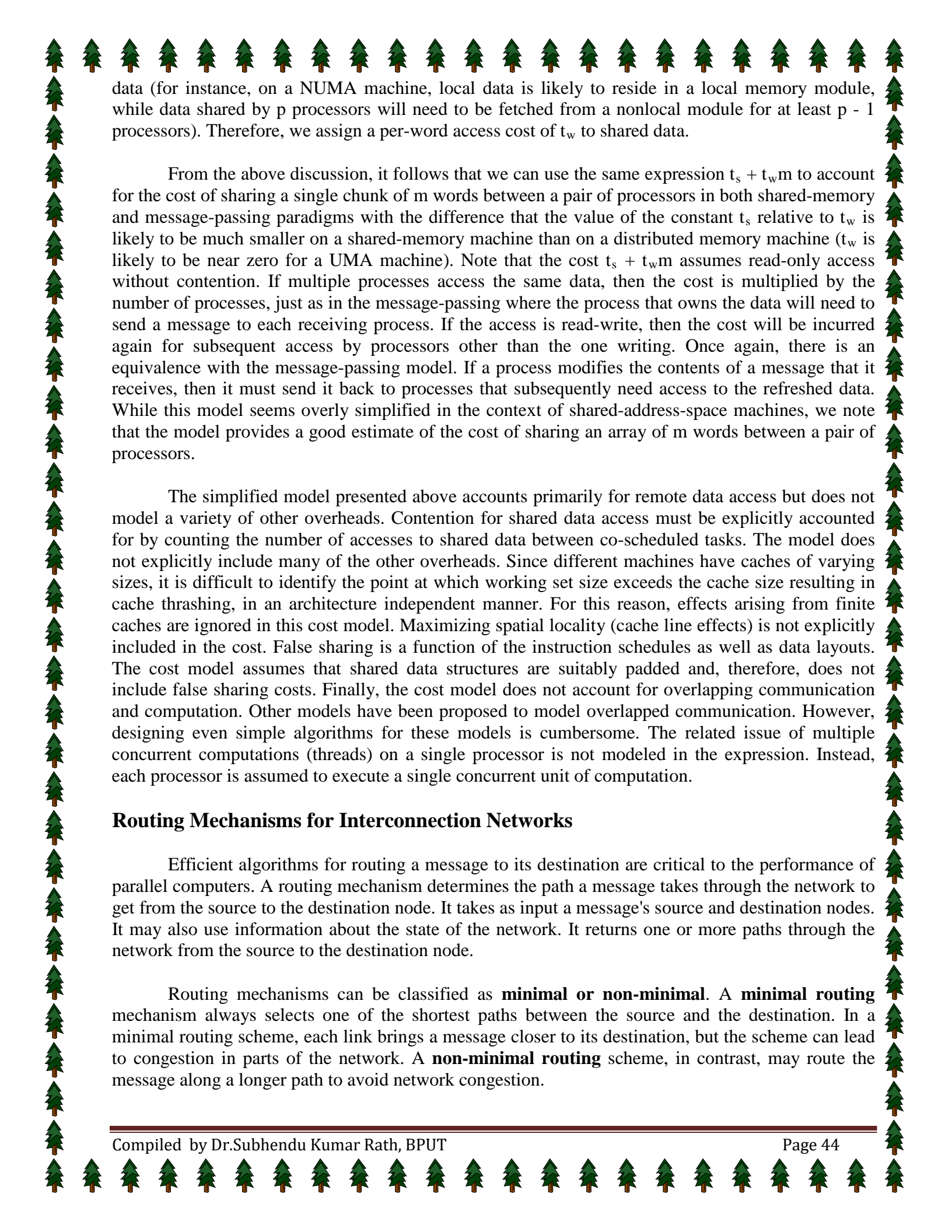


execution schedules for minimizing working set size. While this problem is common to both serial and multiprocessor platforms, the penalty is much higher in the case of multiprocessors since each miss might now involve coherence operations and interprocessor communication.

- **Overheads associated with invalidate and update operations are difficult to quantify.** After a data item has been fetched by a processor into cache, it may be subject to a variety of operations at another processor. For example, in an invalidate protocol, the cache line might be invalidated by a write operation at a remote processor. In this case, the next read operation on the data item must pay a remote access latency cost again. Similarly, the overhead associated with an update protocol might vary significantly depending on the number of copies of a data item. The number of concurrent copies of a data item and the schedule of instruction execution are typically beyond the control of the programmer.
- **Spatial locality is difficult to model.** Since cache lines are generally longer than one word (anywhere from four to 128 words), different words might have different access latencies associated with them even for the first access. Accessing a neighbor of a previously fetched word might be extremely fast, if the cache line has not yet been overwritten. Once again, the programmer has minimal control over this, other than to permute data structures to maximize spatial locality of data reference.
- **Prefetching can play a role in reducing the overhead associated with data access.** Compilers can advance loads and, if sufficient resources exist, the overhead associated with these loads may be completely masked. Since this is a function of the compiler, the underlying program, and availability of resources (registers/cache), it is very difficult to model accurately.
- **False sharing is often an important overhead in many programs.** Two words used by (threads executing on) different processor may reside on the same cache line. This may cause coherence actions and communication overheads, even though none of the data might be shared. The programmer must adequately pad data structures used by various processors to minimize false sharing.
- **Contention in shared accesses is often a major contributing overhead in shared address space machines.** Unfortunately, contention is a function of execution schedule and consequently very difficult to model accurately (independent of the scheduling algorithm). While it is possible to get loose asymptotic estimates by counting the number of shared accesses, such a bound is often not very meaningful.

Any cost model for shared-address-space machines must account for all of these overheads. Building these into a single cost model results in a model that is too cumbersome to design programs for and too specific to individual machines to be generally applicable.

As a first-order model, it is easy to see that accessing a remote word results in a cache line being fetched into the local cache. The time associated with this includes the coherence overheads, network overheads, and memory overheads. The coherence and network overheads are functions of the underlying interconnect (since a coherence operation must be potentially propagated to remote processors and the data item must be fetched). In the absence of knowledge of what coherence operations are associated with a specific access and where the word is coming from, we associate a constant overhead to accessing a cache line of the shared data. For the sake of uniformity with the message-passing model, we refer to this cost as  $t_s$ . Because of various latency-hiding protocols, such as prefetching, implemented in modern processor architectures, we assume that a constant cost of  $t_s$  is associated with initiating access to a contiguous chunk of  $m$  words of shared data, even if  $m$  is greater than the cache line size. We further assume that accessing shared data is costlier than accessing local



data (for instance, on a NUMA machine, local data is likely to reside in a local memory module, while data shared by  $p$  processors will need to be fetched from a nonlocal module for at least  $p - 1$  processors). Therefore, we assign a per-word access cost of  $t_w$  to shared data.

From the above discussion, it follows that we can use the same expression  $t_s + t_w m$  to account for the cost of sharing a single chunk of  $m$  words between a pair of processors in both shared-memory and message-passing paradigms with the difference that the value of the constant  $t_s$  relative to  $t_w$  is likely to be much smaller on a shared-memory machine than on a distributed memory machine ( $t_w$  is likely to be near zero for a UMA machine). Note that the cost  $t_s + t_w m$  assumes read-only access without contention. If multiple processes access the same data, then the cost is multiplied by the number of processes, just as in the message-passing where the process that owns the data will need to send a message to each receiving process. If the access is read-write, then the cost will be incurred again for subsequent access by processors other than the one writing. Once again, there is an equivalence with the message-passing model. If a process modifies the contents of a message that it receives, then it must send it back to processes that subsequently need access to the refreshed data. While this model seems overly simplified in the context of shared-address-space machines, we note that the model provides a good estimate of the cost of sharing an array of  $m$  words between a pair of processors.

The simplified model presented above accounts primarily for remote data access but does not model a variety of other overheads. Contention for shared data access must be explicitly accounted for by counting the number of accesses to shared data between co-scheduled tasks. The model does not explicitly include many of the other overheads. Since different machines have caches of varying sizes, it is difficult to identify the point at which working set size exceeds the cache size resulting in cache thrashing, in an architecture independent manner. For this reason, effects arising from finite caches are ignored in this cost model. Maximizing spatial locality (cache line effects) is not explicitly included in the cost. False sharing is a function of the instruction schedules as well as data layouts. The cost model assumes that shared data structures are suitably padded and, therefore, does not include false sharing costs. Finally, the cost model does not account for overlapping communication and computation. Other models have been proposed to model overlapped communication. However, designing even simple algorithms for these models is cumbersome. The related issue of multiple concurrent computations (threads) on a single processor is not modeled in the expression. Instead, each processor is assumed to execute a single concurrent unit of computation.

## Routing Mechanisms for Interconnection Networks

Efficient algorithms for routing a message to its destination are critical to the performance of parallel computers. A routing mechanism determines the path a message takes through the network to get from the source to the destination node. It takes as input a message's source and destination nodes. It may also use information about the state of the network. It returns one or more paths through the network from the source to the destination node.

Routing mechanisms can be classified as **minimal or non-minimal**. A **minimal routing** mechanism always selects one of the shortest paths between the source and the destination. In a minimal routing scheme, each link brings a message closer to its destination, but the scheme can lead to congestion in parts of the network. A **non-minimal routing** scheme, in contrast, may route the message along a longer path to avoid network congestion.

Routing mechanisms can also be classified on the basis of how they use information regarding the state of the network. A **deterministic routing** scheme determines a unique path for a message, based on its source and destination. It does not use any information regarding the state of the network. Deterministic schemes may result in uneven use of the communication resources in a network. In contrast, an **adaptive routing** scheme uses information regarding the current state of the network to determine the path of the message. Adaptive routing detects congestion in the network and routes messages around it.

One commonly used deterministic minimal routing technique is called **dimension-ordered routing**. Dimension-ordered routing assigns successive channels for traversal by a message based on a numbering scheme determined by the dimension of the channel. The dimension-ordered routing technique for a two-dimensional mesh is called **XY-routing** and that for a hypercube is called **E-cube routing**.

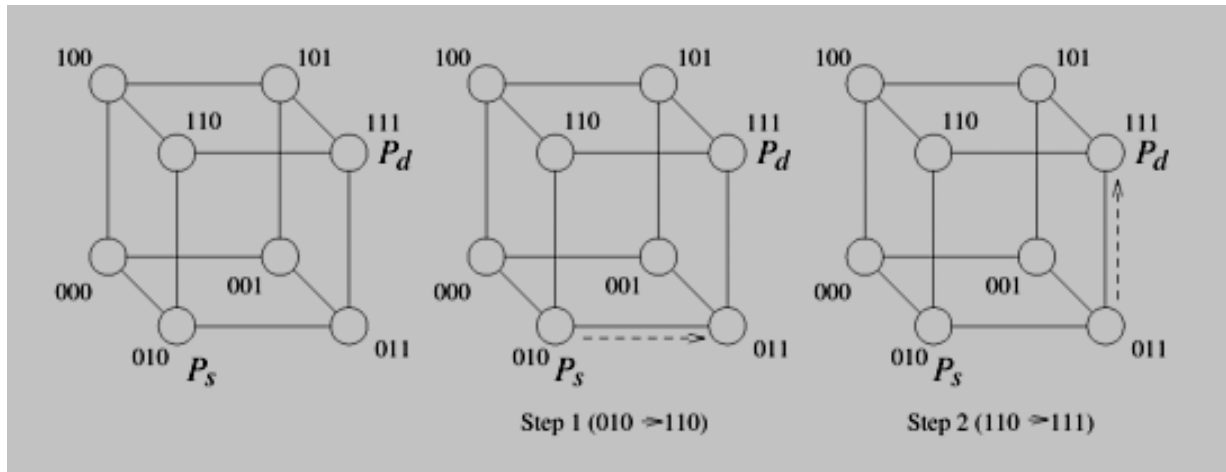
Consider a two-dimensional mesh without wraparound connections. In the XY-routing scheme, a message is sent first along the X dimension until it reaches the column of the destination node and then along the Y dimension until it reaches its destination. Let  $P_{S_y, S_x}$  represent the position of the source node and  $P_{D_y, D_x}$  represent that of the destination node. Any minimal routing scheme should return a path of length  $|S_x - D_x| + |S_y - D_y|$ . Assume that  $D_x \geq S_x$  and  $D_y \geq S_y$ . In the XY-routing scheme, the message is passed through intermediate nodes  $P_{S_y, S_x+1}, P_{S_y, S_x+2}, \dots, P_{S_y, D_x}$  along the X dimension and then through nodes  $P_{S_y+1, D_x}, P_{S_y+2, D_x}, \dots, P_{D_y, D_x}$  along the Y dimension to reach the destination. Note that the length of this path is indeed  $|S_x - D_x| + |S_y - D_y|$ .

E-cube routing for hypercube-connected networks works similarly. Consider a d-dimensional hypercube of p nodes. Let  $P_s$  and  $P_d$  be the labels of the source and destination nodes. We know that the binary representations of these labels are d bits long. Furthermore, the minimum distance between these nodes is given by the number of ones in  $P_s \oplus P_d$  (where  $\oplus$  represents the bitwise exclusive-OR operation). In the E-cube algorithm, node  $P_s$  computes  $P_s \oplus P_d$  and sends the message along dimension k, where k is the position of the least significant nonzero bit in  $P_s \oplus P_d$ . At each intermediate step, node  $P_i$ , which receives the message, computes  $P_i \oplus P_d$  and forwards the message along the dimension corresponding to the least significant nonzero bit. This process continues until the message reaches its destination.

Consider the three-dimensional hypercube shown in the figure. Let  $P_s = 010$  and  $P_d = 111$  represent the source and destination nodes for a message. Node  $P_s$  computes  $010 \oplus 111 = 101$ . In the first step,  $P_s$  forwards the message along the dimension corresponding to the least significant bit to node 011. Node 011 sends the message along the dimension corresponding to the most significant bit

( $011 \oplus 111 = 100$ ). The message reaches node 111, which is the destination of the message.

**Figure . Routing a message from node  $P_s$  (010) to node  $P_d$  (111) in a three-dimensional hypercube using E-cube routing.**



### Mapping Techniques for Graphs

Given two graphs,  $G(V, E)$  and  $G'(V', E')$ , mapping graph  $G$  into graph  $G'$  maps each vertex in the set  $V$  onto a vertex (or a set of vertices) in set  $V'$  and each edge in the set  $E$  onto an edge (or a set of edges) in  $E'$ . When mapping graph  $G(V, E)$  into  $G'(V', E')$ , three parameters are important.

First, it is possible that more than one edge in  $E$  is mapped onto a single edge in  $E'$ . The maximum number of edges mapped onto any edge in  $E'$  is called the **congestion** of the mapping.

Second, an edge in  $E$  may be mapped onto multiple contiguous edges in  $E'$ . This is significant because traffic on the corresponding communication link must traverse more than one link, possibly contributing to congestion on the network. The maximum number of links in  $E'$  that any edge in  $E$  is mapped onto is called the **dilation** of the mapping.

Third, the sets  $V$  and  $V'$  may contain different numbers of vertices. In this case, a node in  $V$  corresponds to more than one node in  $V'$ . The ratio of the number of nodes in the set  $V'$  to that in set  $V$  is called the **expansion** of the mapping. In the context of process-processor mapping, we want the expansion of the mapping to be identical to the ratio of virtual and physical processors.

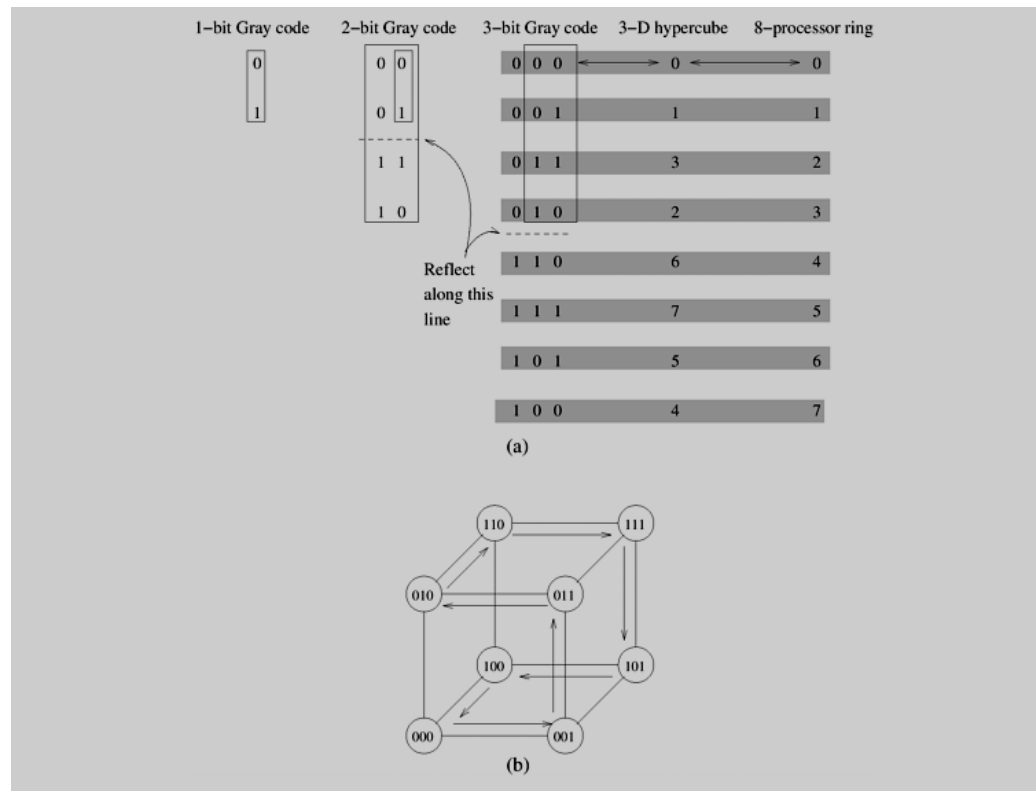
### Embedding a Linear Array into a Hypercube

A linear array (or a ring) composed of  $2^d$  nodes (labeled 0 through  $2^d - 1$ ) can be embedded into a  $d$ -dimensional hypercube by mapping node  $i$  of the linear array onto node  $G(i, d)$  of the hypercube. The function  $G(i, x)$  is defined as follows:

$$\begin{aligned}
 G(0, 1) &= 0 \\
 G(1, 1) &= 1 \\
 G(i, x + 1) &= \begin{cases} G(i, x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
 \end{aligned}$$

The function  $G$  is called the binary reflected Gray code (RGC). The entry  $G(i, d)$  denotes the  $i$ th entry in the sequence of Gray codes of  $d$  bits. Gray codes of  $d + 1$  bits are derived from a table of Gray codes of  $d$  bits by reflecting the table and prefixing the reflected entries with a 1 and the original entries with a 0.

Figure . (a) A three-bit reflected Gray code ring; and (b) its embedding into a three-dimensional hypercube.



A careful look at the Gray code table reveals that two adjoining entries ( $G(i, d)$  and  $G(i + 1, d)$ ) differ from each other at only one bit position. Since node  $i$  in the linear array is mapped to node  $G(i, d)$ , and node  $i + 1$  is mapped to  $G(i + 1, d)$ , there is a direct link in the hypercube that corresponds to each direct link in the linear array. (Recall that two nodes whose labels differ at only one bit position have a direct link in a hypercube.) Therefore, the mapping specified by the function  $G$  has a dilation of one and a congestion of one.

### Embedding a Mesh into a Hypercube

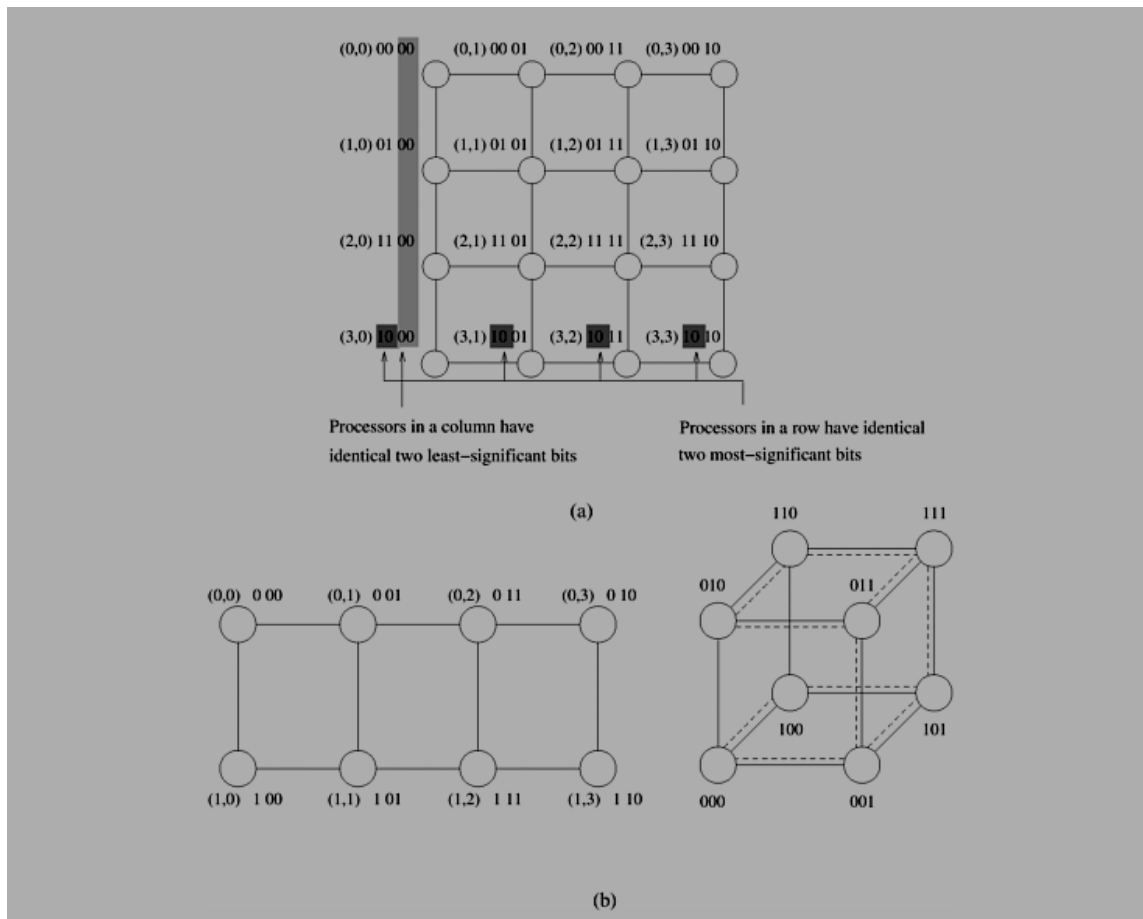
Embedding a mesh into a hypercube is a natural extension of embedding a ring into a hypercube. We can embed a  $2^r \times 2^s$  wraparound mesh into a  $2^{r+s}$ -node hypercube by mapping node  $(i, j)$  of the mesh onto node  $G(i, r - 1) || G(j, s - 1)$  of the hypercube (where  $||$  denotes concatenation of the two Gray codes). Note that immediate neighbors in the mesh are mapped to hypercube nodes whose labels differ in exactly one bit position. Therefore, this mapping has a dilation of one and a congestion of one.

For example, consider embedding a  $2 \times 4$  mesh into an eight-node hypercube. The values of  $r$  and  $s$  are 1 and 2, respectively. Node  $(i, j)$  of the mesh is mapped to node  $G(i, 1) || G(j, 2)$  of the hypercube. Therefore, node  $(0, 0)$  of the mesh is mapped to node 000 of the hypercube, because  $G(0,$



1) is 0 and  $G(0, 2)$  is 00; concatenating the two yields the label 000 for the hypercube node. Similarly, node (0, 1) of the mesh is mapped to node 001 of the hypercube, and so on.

**Figure (a)** A 4 x 4 mesh illustrating the mapping of mesh nodes to the nodes in a four-dimensional hypercube; and **(b)** a 2 x 4 mesh embedded into a three-dimensional hypercube.



This mapping of a mesh into a hypercube has certain useful properties. All nodes in the same row of the mesh are mapped to hypercube nodes whose labels have  $r$  identical most significant bits. We know that fixing any  $r$  bits in the node label of an  $(r + s)$ -dimensional hypercube yields a subcube of dimension  $s$  with  $2^s$  nodes. Since each mesh node is mapped onto a unique node in the hypercube, and each row in the mesh has  $2^s$  nodes, every row in the mesh is mapped to a distinct subcube in the hypercube. Similarly, each column in the mesh is mapped to a distinct subcube in the hypercube.

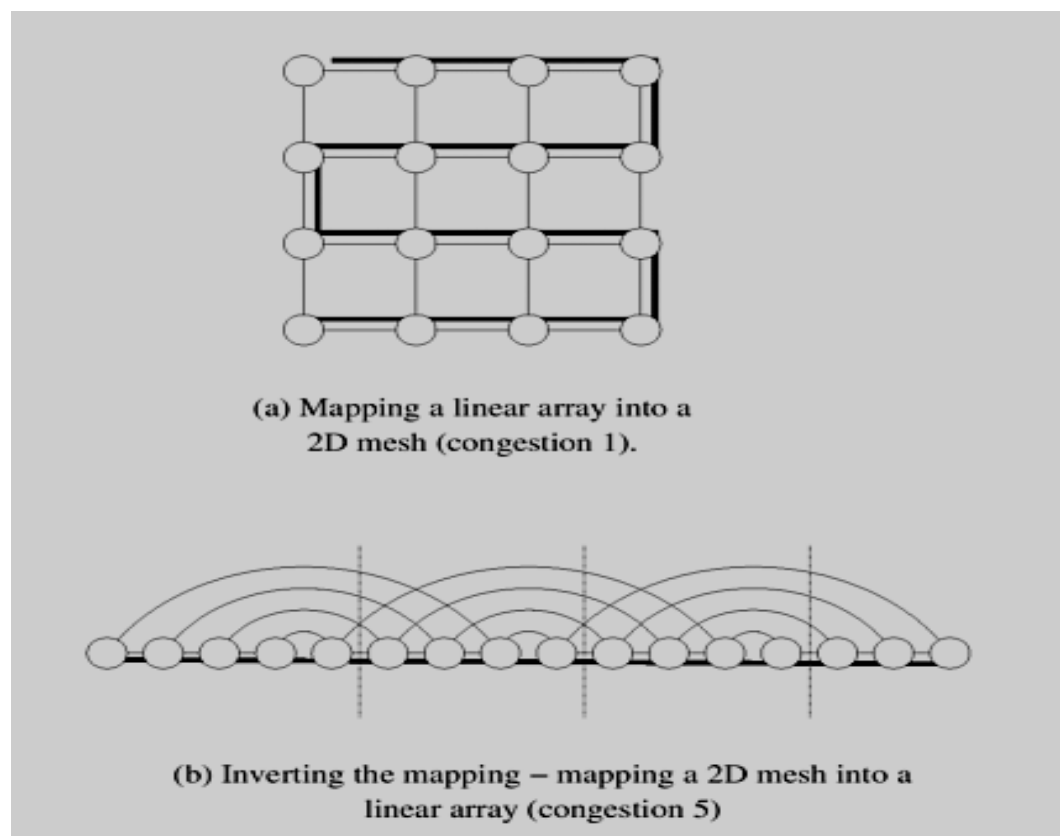
### Embedding a Mesh into a Linear Array

We have, up until this point, considered embeddings of sparser networks into denser networks. A 2-D mesh has  $2 \times p$  links. In contrast, a  $p$ -node linear array has  $p$  links. Consequently, there must be a congestion associated with this mapping.

Consider first the mapping of a linear array into a mesh. We assume that neither the mesh nor the linear array has wraparound connections. Here, the solid lines correspond to links in the linear

array and normal lines to links in the mesh. It is easy to see from figure that a congestion-one, dilation-one mapping of a linear array to a mesh is possible.

Figure (a) Embedding a 16 node linear array into a 2-D mesh; and (b) the inverse of the mapping. Solid lines correspond to links in the linear array and normal lines to links in the mesh.



Consider now the inverse of this mapping, i.e., we are given a mesh and we map vertices of the mesh to those in a linear array using the inverse of the same mapping function. As before, the solid lines correspond to edges in the linear array and normal lines to edges in the mesh. As is evident from the figure, the congestion of the mapping in this case is five – i.e., no solid line carries more than five normal lines. In general, it is easy to show that the congestion of this (inverse) mapping is  $\sqrt{p} + 1$  for a general  $p$ -node mapping (one for each of the  $\sqrt{p}$  edges to the next row, and one additional edge).

While this is a simple mapping, the question at this point is whether we can do better. To answer this question, we use the bisection width of the two networks. We know that the bisection width of a 2-D mesh without wraparound links is  $\sqrt{p}$ , and that of a linear array is 1. Assume that the best mapping of a 2-D mesh into a linear array has a congestion of  $r$ . This implies that if we take the linear array and cut it in half (at the middle), we will cut only one linear array link, or no more than  $r$  mesh links. We claim that  $r$  must be at least equal to the bisection width of the mesh. This follows from the fact that an equi-partition of the linear array into two also partitions the mesh into two. Therefore, at least  $\sqrt{p}$  mesh links must cross the partition, by definition of bisection width. Consequently, the one linear array link connecting the two halves must carry at least  $\sqrt{p}$  mesh links.

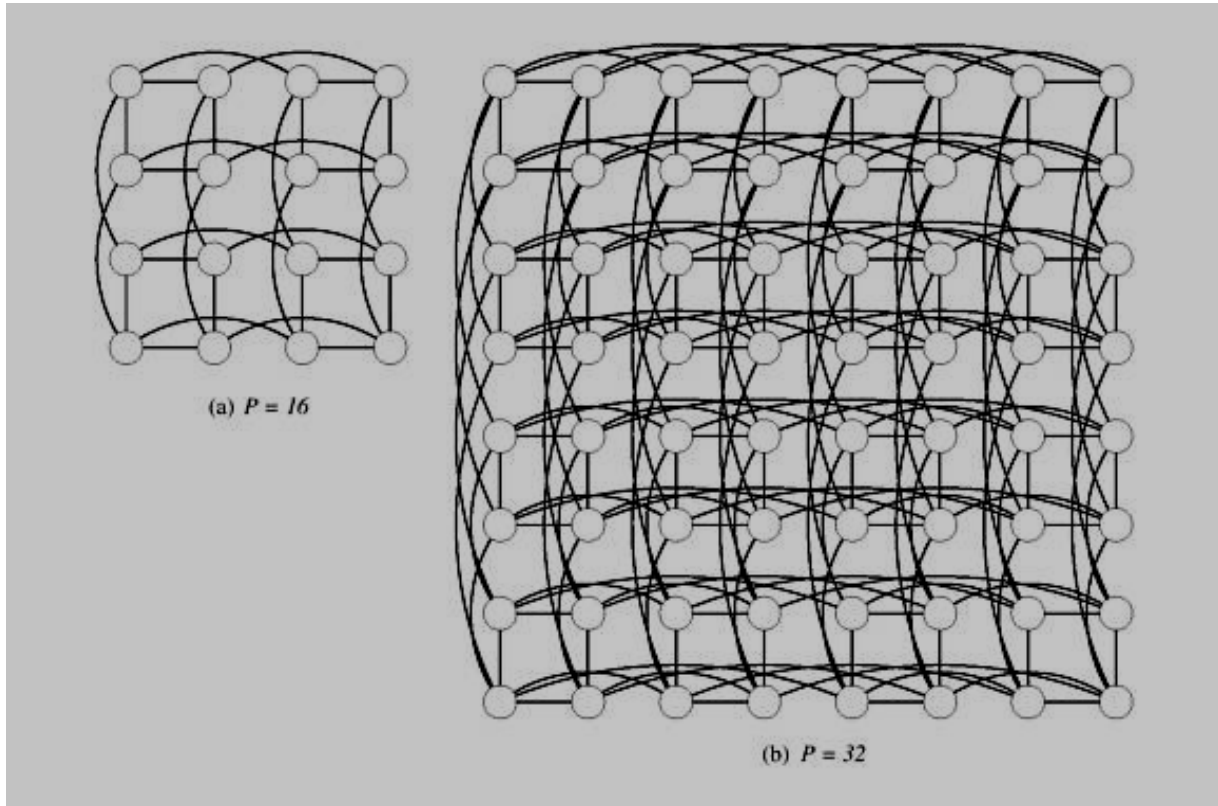
Therefore, the congestion of any mapping is lower bounded by  $\sqrt{p}$ . The lower bound established above has a more general applicability when mapping denser networks to sparser ones. One may reasonably believe that the lower bound on congestion of a mapping of network S with x links into network Q with y links is x/y. In the case of the mapping from a mesh to a linear array, this would be  $2p/p$ , or 2. However, this lower bound is overly conservative. A tighter lower bound is in fact possible by examining the bisection width of the two networks.

### Embedding a Hypercube into a 2-D Mesh

Consider the embedding of a p-node hypercube into a p-node 2-D mesh. For the sake of convenience, we assume that p is an even power of two. In this scenario, it is possible to visualize the hypercube as  $\sqrt{p}$  subcubes, each with  $\sqrt{p}$  nodes. We do this as follows: let  $d = \log p$  be the dimension of the hypercube. From our assumption, we know that d is even. We take the  $d/2$  least significant bits and use them to define individual subcubes of  $\sqrt{p}$  nodes. For example, in the case of a 4D hypercube, we use the lower two bits to define the subcubes as (0000, 0001, 0011, 0010), (0100, 0101, 0111, 0110), (1100, 1101, 1111, 1110), and (1000, 1001, 1011, 1010). Note at this point that if we fix the  $d/2$  least significant bits across all of these subcubes, we will have another subcube as defined by the  $d/2$  most significant bits. For example, if we fix the lower two bits across the subcubes to 10, we get the nodes (0010, 0110, 1110, 1010). The reader can verify that this corresponds to a 2-D subcube.

The mapping from a hypercube to a mesh can now be defined as follows: each subcube is mapped to a  $\sqrt{p}$  node row of the mesh. We do this by simply inverting the linear-array to hypercube mapping. The bisection width of the  $\sqrt{p}$  node hypercube is  $\sqrt{p}/2$ . The corresponding bisection width of a  $\sqrt{p}$  node row is 1. Therefore the congestion of this subcube-to-row mapping is  $\sqrt{p}/2$  (at the edge that connects the two halves of the row). This is illustrated for the cases of  $p = 16$  and  $p = 32$  in figure (a) and (b). In this fashion, we can map each subcube to a different row in the mesh. Note that while we have computed the congestion resulting from the subcube-to-row mapping, we have not addressed the congestion resulting from the column mapping. We map the hypercube nodes into the mesh in such a way that nodes with identical  $d/2$  least significant bits in the hypercube are mapped to the same column. This results in a subcube-to-column mapping, where each subcube/column has  $\sqrt{p}$  nodes. Using the same argument as in the case of subcube-to-row mapping, this results in a congestion of  $\sqrt{p}/2$ . Since the congestion from the row and column mappings affects disjoint sets of edges, the total congestion of this mapping is  $\sqrt{p}/2$ .

Figure. Embedding a hypercube into a 2-D mesh.



Since the bisection width of a hypercube is  $p/2$  and that of a mesh is  $\sqrt{p}$ , the lower bound on congestion is the ratio of these, i.e.,  $\sqrt{p}/2$ . We notice that our mapping yields this lower bound on congestion.

## Principles of Parallel Algorithm Design

Algorithm development is a critical component of problem solving using computers. A sequential algorithm is essentially a recipe or a sequence of basic steps for solving a given problem using a serial computer. Similarly, a parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors. However, specifying a parallel algorithm involves more than just specifying the steps. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify sets of steps that can be executed simultaneously. This is essential for obtaining any performance benefit from the use of a parallel computer. In practice, specifying a nontrivial parallel algorithm may include some or all of the following:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Typically, there are several choices for each of the above steps, but usually, relatively few combinations of choices lead to a parallel algorithm that yields performance commensurate with the computational and storage resources employed to solve the problem. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms.

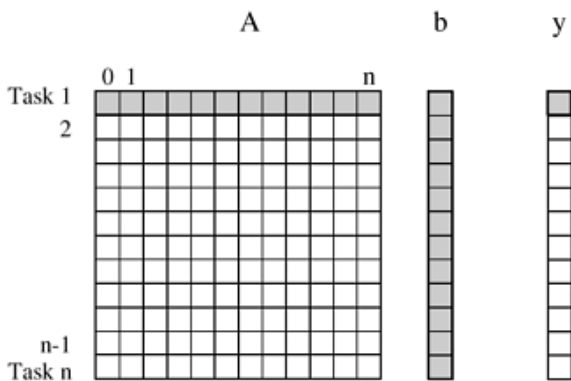
### Decomposition, Tasks, and Dependency Graphs

The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called **decomposition**. **Tasks** are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem. Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation. The tasks into which a problem is decomposed may not all be of the same size.

#### Example : Dense matrix-vector multiplication

Consider the multiplication of a dense  $n \times n$  matrix  $A$  with a vector  $b$  to yield another vector  $y$ . The  $i$ th element  $y[i]$  of the product vector is the dot-product of the  $i$ th row of  $A$  with the input vector  $b$ ; i.e.,  $y[i] = \sum_{j=1}^n A[i, j].b[j]$ . As shown in figure-1 the computation of each  $y[i]$  can be regarded as a task. Alternatively, as shown in figure-2 the computation could be decomposed into fewer, say four,

tasks where each task computes roughly  $n/4$  of the entries of the vector  $y$ . **Figure-1. Decomposition of dense matrix-vector multiplication into  $n$  tasks, where  $n$  is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.**



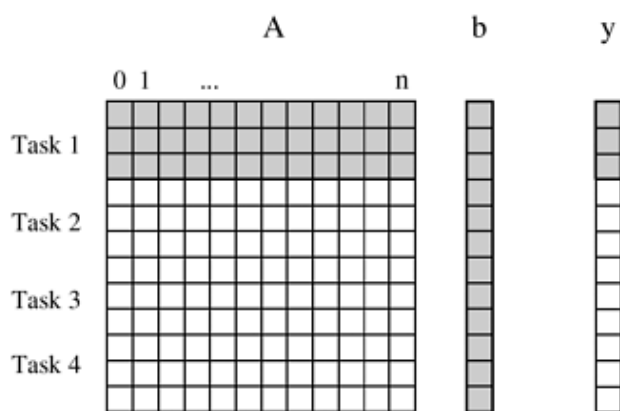
Note that all tasks in Figure-1 are independent and can be performed all together or in any sequence. However, in general, some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution. An abstraction used to express such dependencies among tasks and their relative order of execution is known as a task-dependency graph. A **task-dependency graph** is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them. The task corresponding to a node can be executed when all tasks

connected to this node by incoming edges have completed. Note that task-dependency graphs can be disconnected and the edge-set of a task-dependency graph can be empty. This is the case for matrix-vector multiplication, where each task computes a subset of the entries of the product vector.

### Granularity, Concurrency, and Task-Interaction

The number and size of tasks into which a problem is decomposed determines the **granularity** of the decomposition. Decomposition into a large number of small tasks is called **fine-grained** and decomposition into a small number of large tasks is called **coarse-grained**. For example, the decomposition for matrix-vector multiplication shown in Figure-1 would usually be considered fine-grained because each of a large number of tasks performs a single dot-product. Figure-2 shows a coarse-grained decomposition of the same problem into four tasks, where each task computes  $n/4$  of the entries of the output vector of length  $n$ .

Figure-2. Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



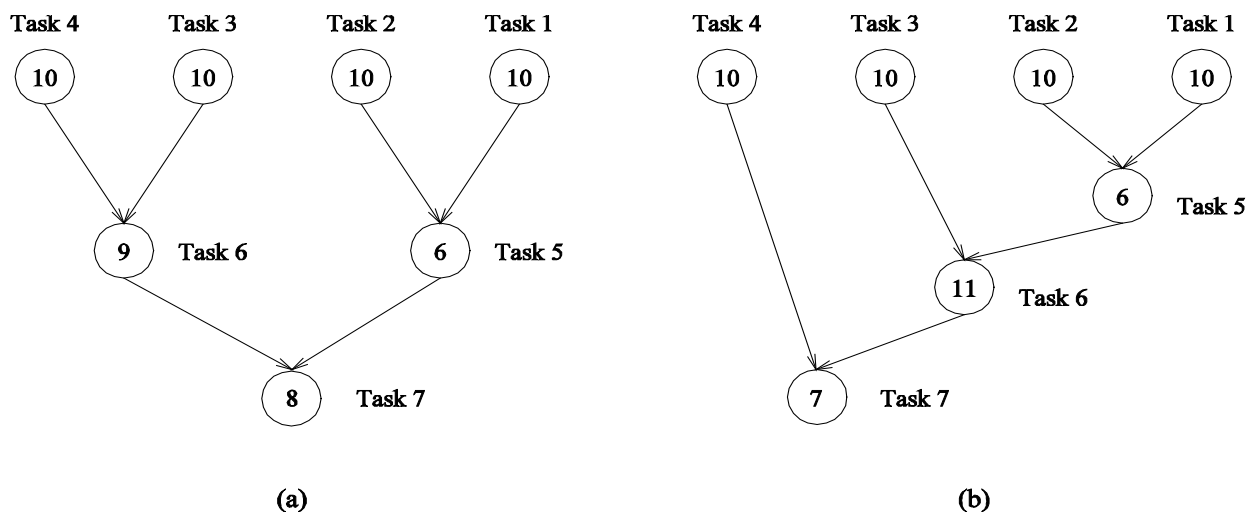
A concept related to granularity is that of degree of concurrency. The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its **maximum degree of concurrency**. In most cases, the maximum degree of concurrency is less than the total number of tasks due to dependencies among the tasks.

A more useful indicator of a parallel program's performance is the average degree of concurrency, which is the average number of tasks that can run concurrently over the entire duration of execution of the program.

Both the maximum and the average degrees of concurrency usually increase as the granularity of tasks becomes smaller (finer). For example, the decomposition of matrix-vector multiplication shown in Figure-1 has a fairly small granularity and a large degree of concurrency. The decomposition for the same problem shown in Figure-2 has a larger granularity and a smaller degree of concurrency.

A feature of a task-dependency graph that determines the average degree of concurrency for a given granularity is its **critical path**. In a task-dependency graph, let us refer to the nodes with no incoming edges by **start nodes** and the nodes with no outgoing edges by **finish nodes**. The longest directed

path between any pair of start and finish nodes is known as the **critical path**. The sum of the weights of nodes along this path is known as the **critical path length**, where the weight of a node is the size or the amount of work associated with the corresponding task. The ratio of the total amount of work to the critical-path length is the average degree of concurrency. Therefore, a shorter critical path favors a higher degree of concurrency. For example, the critical path length is 27 in the task-dependency graph shown in figure (a) and is 34 in the task-dependency graph shown in figure (b). Since the total amount of work required to solve the problems using the two decompositions is 63 and 64, respectively, the average degree of concurrency of the two task-dependency graphs is 2.33 and 1.88, respectively.




Although it may appear that the time required to solve a problem can be reduced simply by increasing the granularity of decomposition and utilizing the resulting concurrency to perform more and more tasks in parallel, this is not the case in most practical scenarios. Usually, there is an inherent bound on how fine-grained a decomposition a problem permits.

### Processes and Mapping

The tasks, into which a problem is decomposed, run on physical processors. However, for reasons that we shall soon discuss, we will use the term process in this chapter to refer to a processing or computing agent that performs tasks. In this context, the term process does not adhere to the rigorous operating system definition of a process. Instead, it is an abstract entity that uses the code and data corresponding to a task to produce the output of that task within a finite amount of time after the task is activated by the parallel program. During this time, in addition to performing computations, a process may synchronize or communicate with other processes, if needed. In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks. The mechanism by which tasks are assigned to processes for execution is called **mapping**.

The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good mapping for a parallel algorithm. A good mapping should seek to maximize the use of concurrency by mapping independent tasks onto different processes, it should seek to minimize the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become executable, and it should seek to minimize interaction among processes by mapping tasks with a high degree of mutual



interaction onto the same process. In most nontrivial parallel algorithms, these tend to be conflicting goals.

### Processes versus Processors

In the context of parallel algorithm design, **processes** are logical computing agents that perform tasks. **Processors** are the hardware units that physically perform computations. In this text, we choose to express parallel algorithms and programs in terms of processes. In most cases, when we refer to processes in the context of a parallel algorithm, there is a one-to-one correspondence between processes and processors and it is appropriate to assume that there are as many processes as the number of physical CPUs on the parallel computer. However, sometimes a higher level of abstraction may be required to express a parallel algorithm, especially if it is a complex algorithm with multiple stages or with different forms of parallelism.

Treating processes and processors separately is also useful when designing parallel programs for hardware that supports multiple programming paradigms. For instance, consider a parallel computer that consists of multiple computing nodes that communicate with each other via message passing. Now each of these nodes could be a shared-address-space module with multiple CPUs. Consider implementing matrix multiplication on such a parallel computer. The best way to design a parallel algorithm is to do so in two stages. First, develop a decomposition and mapping strategy suitable for the message-passing paradigm and use this to exploit parallelism among the nodes. Each task that the original matrix multiplication problem decomposes into is a matrix multiplication computation itself. The next step is to develop a decomposition and mapping strategy suitable for the shared-memory paradigm and use this to implement each task on the multiple CPUs of a node.

### Decomposition Techniques

One of the fundamental steps that we need to undertake to solve a problem in parallel is to split the computations to be performed into a set of tasks for concurrent execution defined by the task-dependency graph.

These techniques are broadly classified as **recursive decomposition**, **data-decomposition**, **exploratory decomposition**, and **speculative decomposition**. The recursive- and data-decomposition techniques are relatively general purpose as they can be used to decompose a wide variety of problems. On the other hand, speculative- and exploratory-decomposition techniques are more of a special purpose nature because they apply to specific classes of problems.

### Recursive Decomposition

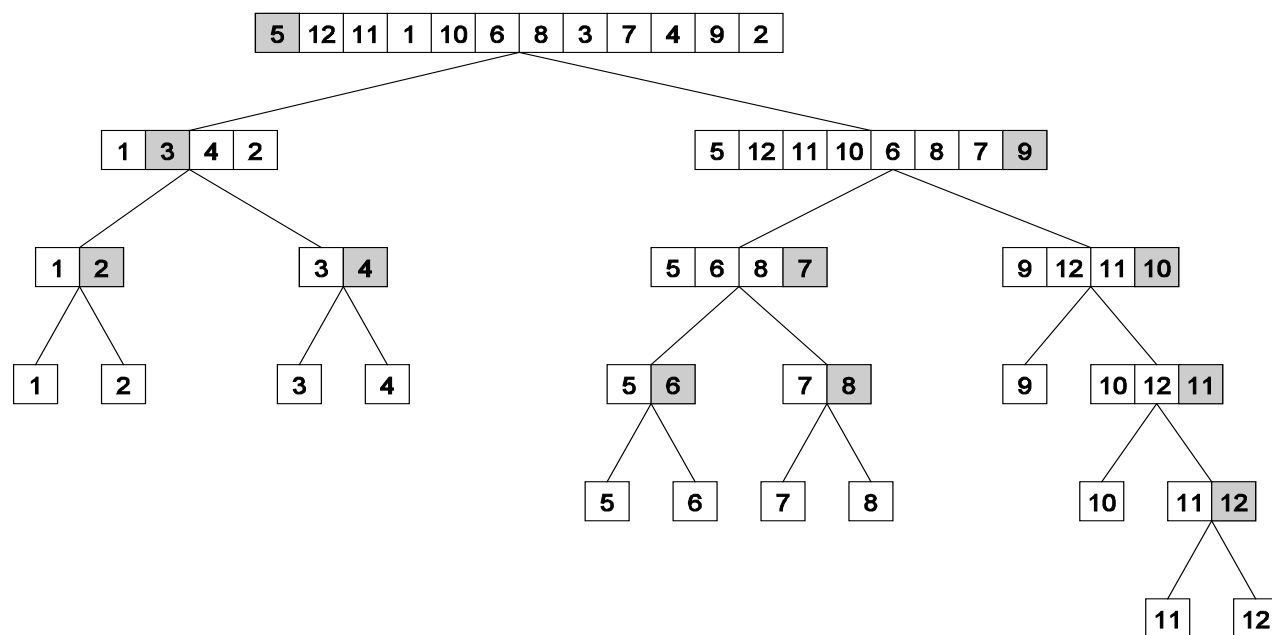
Recursive decomposition is a method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.

Example : Quicksort



Consider the problem of sorting a sequence  $A$  of  $n$  elements using the commonly used quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element  $x$  and then partitions the sequence  $A$  into two subsequences  $A_0$  and  $A_1$  such that all the elements in  $A_0$  are smaller than  $x$  and all the elements in  $A_1$  are greater than or equal to  $x$ . This partitioning step forms the divide step of the algorithm. Each one of the subsequences  $A_0$  and  $A_1$  is sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. The recursion terminates when each subsequence contains only a single element.

Figure : The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.



In Figure, we define a task as the work of partitioning a given subsequence. Initially, there is only one sequence (i.e., the root of the tree), and we can use only a single process to partition it. The completion of the root task results in two subsequences ( $A_0$  and  $A_1$ , corresponding to the two nodes at the first level of the tree) and each one can be partitioned in parallel. Similarly, the concurrency continues to increase as we move down the tree.

### Data Decomposition

Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps.

In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication) or are chosen from a small set of operations (e.g., LU factorization)

The partitioning of data can be performed in many possible ways as discussed next. In general, one must explore and evaluate all possible ways of partitioning the data and determine which one yields a natural and efficient computational decomposition.

**Partitioning Output Data :** In many computations, each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output.

Example : Matrix multiplication

Consider the problem of multiplying two  $n \times n$  matrices  $A$  and  $B$  to yield a matrix  $C$ . Figure shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting each dimension of the matrix into half. The four submatrices of  $C$ , roughly of size  $n/2 \times n/2$  each, are then independently computed by four tasks as the sums of the appropriate products of submatrices of  $A$  and  $B$ .

Figure : (a) Partitioning of input and output matrices into  $2 \times 2$  submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Most matrix algorithms, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations. In such a formulation, the matrix is viewed as composed of blocks or submatrices and the scalar arithmetic operations on its elements are replaced by the equivalent matrix operations on the blocks.

**Partitioning Input Data :** Partitioning of output data can be performed only if each output can be naturally computed as a function of the input. In many algorithms, it is not possible or desirable to partition the output data. For example, while finding the minimum, maximum, or the sum of a set of numbers, the output is a single unknown value. In a sorting algorithm, the individual elements of the output cannot be efficiently determined in isolation. In such cases, it is sometimes possible to partition the input data, and then use this partitioning to induce concurrency. A task is created for each partition of the input data and this task performs as much computation as possible using these local data. Note that the solutions to tasks induced by input partitions may not directly solve the original problem. In such cases, a follow-up computation is needed to combine the results. For example, while finding the sum of a sequence of  $N$  numbers using  $p$  processes ( $N > p$ ), we can partition the input into  $p$  subsets of nearly equal sizes. Each task then computes the sum of the numbers in one of the subsets. Finally, the  $p$  partial results can be added up to yield the final result.

**Partitioning both Input and Output Data :** In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency.

**Partitioning Intermediate Data :** Algorithms are often structured as multi-stage computations such that the output of one stage is the input to the subsequent stage. A decomposition of such an algorithm can be derived by partitioning the input or the output data of an intermediate stage of the algorithm. Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data. Often, the intermediate data are not generated explicitly in the serial algorithm for solving the problem and some restructuring of the original algorithm may be required to use intermediate data partitioning to induce a decomposition.

**The Owner-Computes Rule :** A decomposition based on partitioning output or input data is also widely referred to as the **owner-computes rule**. The idea behind this rule is that each partition performs all the computations involving data that it owns. Depending on the nature of the data or the type of data-partitioning, the owner-computes rule may mean different things. For instance, when we assign partitions of the input data to tasks, then the owner-computes rule means that a task performs all the computations that can be done using these data. On the other hand, if we partition the output data, then the owner-computes rule means that a task computes all the data in the partition assigned to it.

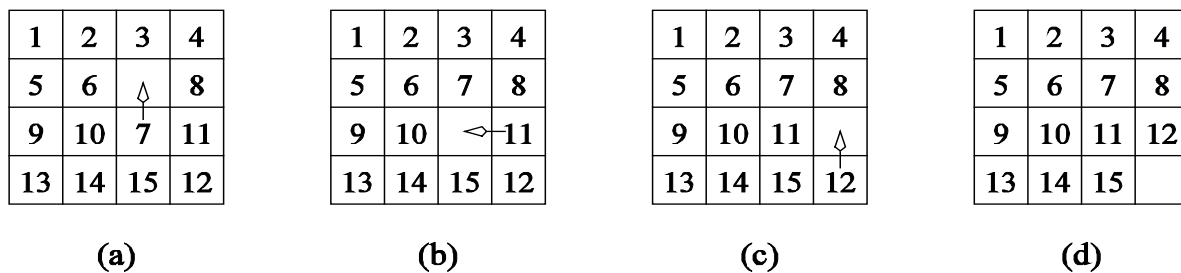
### Exploratory Decomposition

Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found. For an example of exploratory decomposition, consider the 15-puzzle problem.

Example : The 15-puzzle problem

The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration. The Figure illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration.

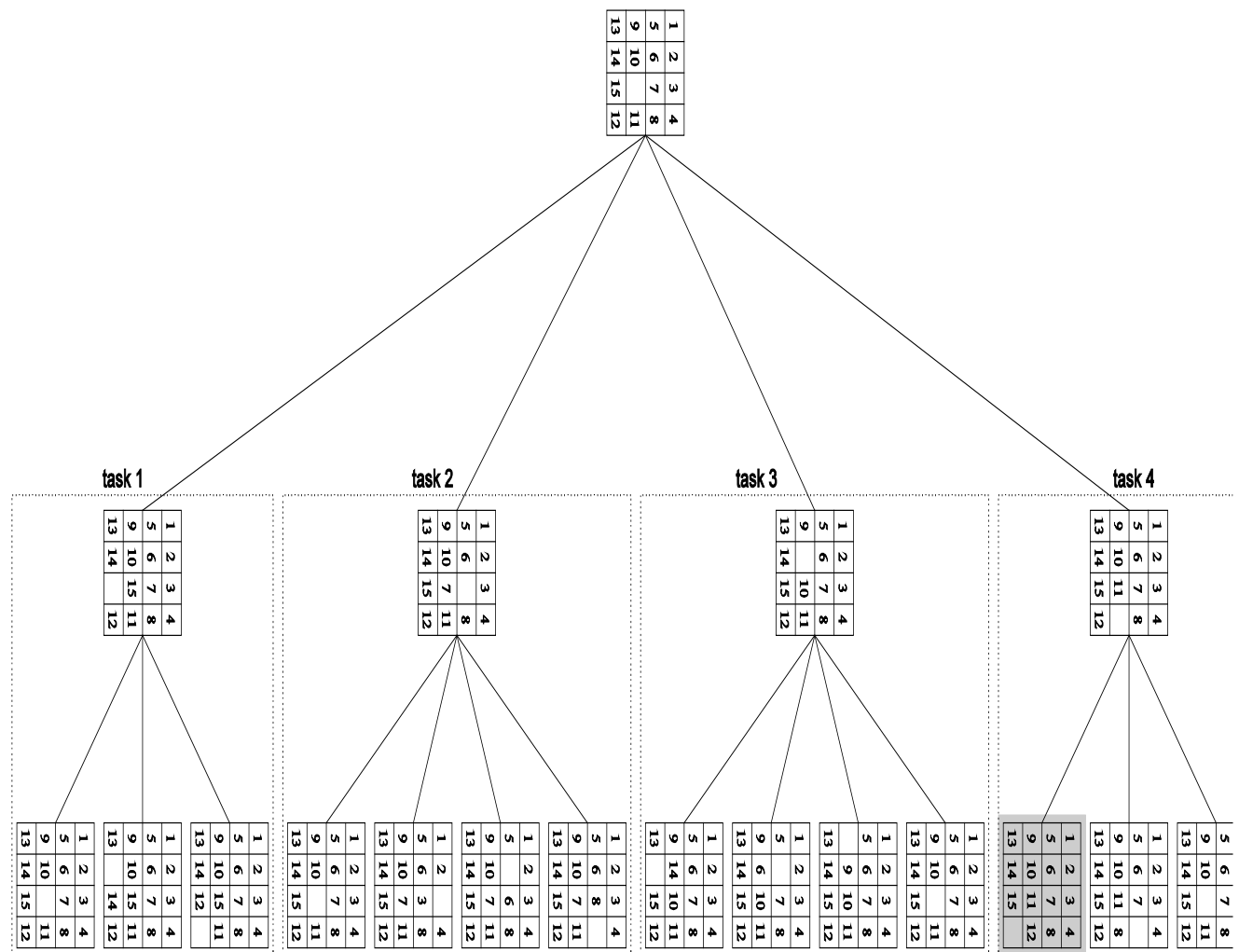
Figure. A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

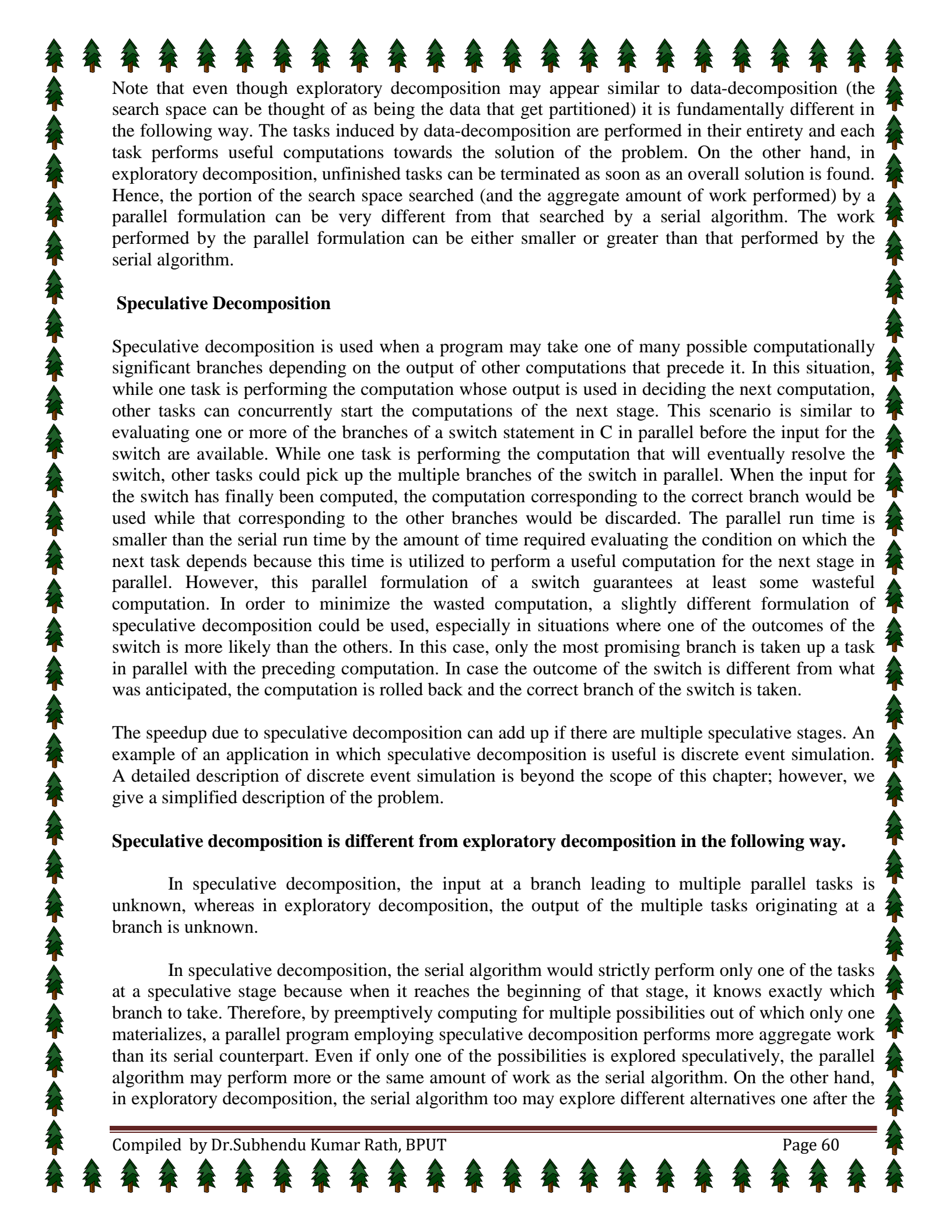


The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successor configurations are generated. A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors. The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated configurations to the final configuration. Since one of these newly generated configurations must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution. The configuration space generated by the tree search is often referred to as a state space graph. Each node of the graph is a configuration and each edge of the graph connects configurations that can be reached from one another by a single move of a tile.

One method for solving this problem in parallel is as follows. First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes (i.e., configurations of the 15-puzzle). Now each node is assigned to a task to explore further until at least one of them finds a solution. As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches. The following figure illustrates one such decomposition into four tasks in which task 4 finds the solution.

Figure : The states generated by an instance of the 15-puzzle problem.





Note that even though exploratory decomposition may appear similar to data-decomposition (the search space can be thought of as being the data that get partitioned) it is fundamentally different in the following way. The tasks induced by data-decomposition are performed in their entirety and each task performs useful computations towards the solution of the problem. On the other hand, in exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found. Hence, the portion of the search space searched (and the aggregate amount of work performed) by a parallel formulation can be very different from that searched by a serial algorithm. The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm.

### **Speculative Decomposition**

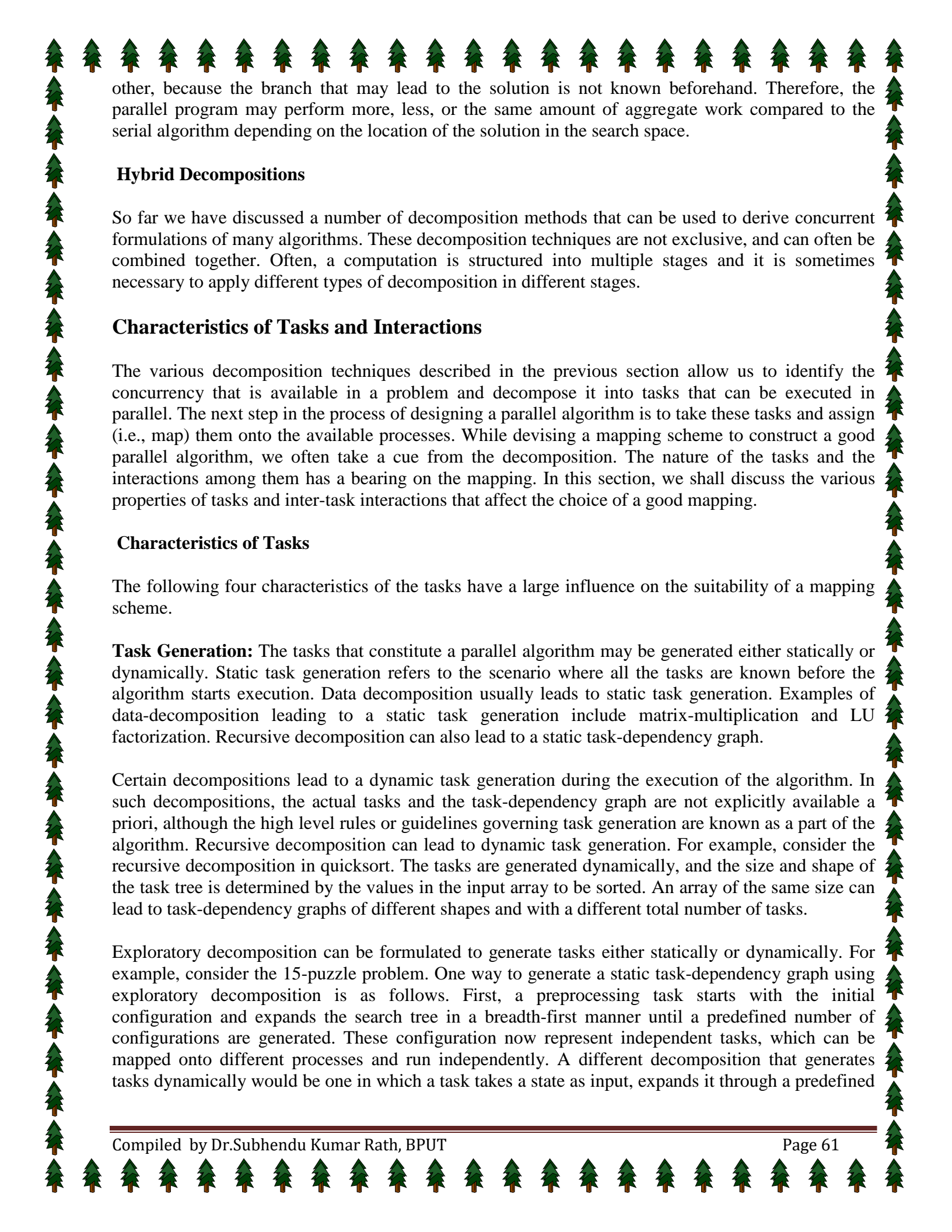
Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage. This scenario is similar to evaluating one or more of the branches of a switch statement in C in parallel before the input for the switch are available. While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel. When the input for the switch has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded. The parallel run time is smaller than the serial run time by the amount of time required evaluating the condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel. However, this parallel formulation of a switch guarantees at least some wasteful computation. In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others. In this case, only the most promising branch is taken up a task in parallel with the preceding computation. In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.

The speedup due to speculative decomposition can add up if there are multiple speculative stages. An example of an application in which speculative decomposition is useful is discrete event simulation. A detailed description of discrete event simulation is beyond the scope of this chapter; however, we give a simplified description of the problem.

### **Speculative decomposition is different from exploratory decomposition in the following way.**

In speculative decomposition, the input at a branch leading to multiple parallel tasks is unknown, whereas in exploratory decomposition, the output of the multiple tasks originating at a branch is unknown.

In speculative decomposition, the serial algorithm would strictly perform only one of the tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take. Therefore, by preemptively computing for multiple possibilities out of which only one materializes, a parallel program employing speculative decomposition performs more aggregate work than its serial counterpart. Even if only one of the possibilities is explored speculatively, the parallel algorithm may perform more or the same amount of work as the serial algorithm. On the other hand, in exploratory decomposition, the serial algorithm too may explore different alternatives one after the



other, because the branch that may lead to the solution is not known beforehand. Therefore, the parallel program may perform more, less, or the same amount of aggregate work compared to the serial algorithm depending on the location of the solution in the search space.

## Hybrid Decompositions

So far we have discussed a number of decomposition methods that can be used to derive concurrent formulations of many algorithms. These decomposition techniques are not exclusive, and can often be combined together. Often, a computation is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages.

## Characteristics of Tasks and Interactions

The various decomposition techniques described in the previous section allow us to identify the concurrency that is available in a problem and decompose it into tasks that can be executed in parallel. The next step in the process of designing a parallel algorithm is to take these tasks and assign (i.e., map) them onto the available processes. While devising a mapping scheme to construct a good parallel algorithm, we often take a cue from the decomposition. The nature of the tasks and the interactions among them has a bearing on the mapping. In this section, we shall discuss the various properties of tasks and inter-task interactions that affect the choice of a good mapping.

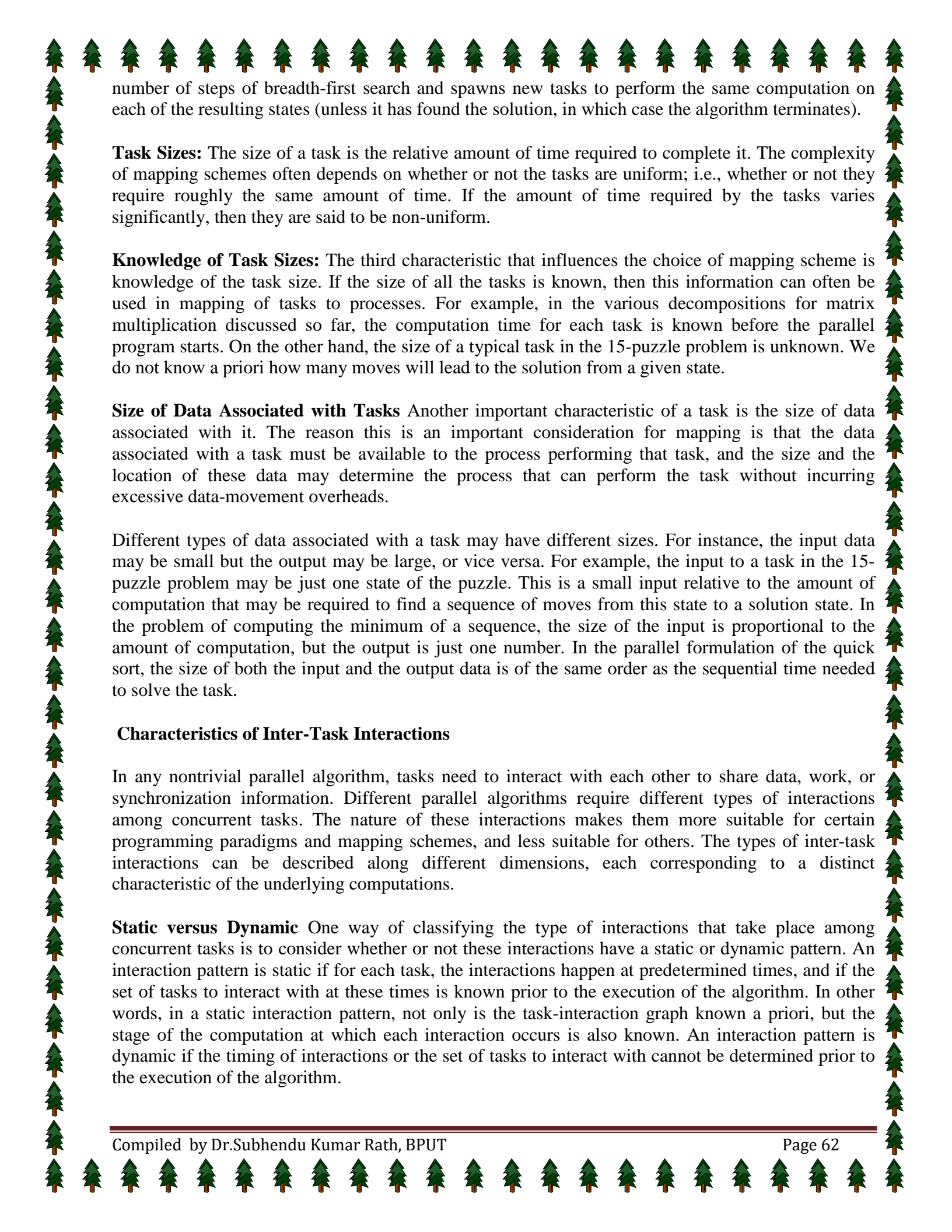
### Characteristics of Tasks

The following four characteristics of the tasks have a large influence on the suitability of a mapping scheme.

**Task Generation:** The tasks that constitute a parallel algorithm may be generated either statically or dynamically. Static task generation refers to the scenario where all the tasks are known before the algorithm starts execution. Data decomposition usually leads to static task generation. Examples of data-decomposition leading to a static task generation include matrix-multiplication and LU factorization. Recursive decomposition can also lead to a static task-dependency graph.

Certain decompositions lead to a dynamic task generation during the execution of the algorithm. In such decompositions, the actual tasks and the task-dependency graph are not explicitly available a priori, although the high level rules or guidelines governing task generation are known as a part of the algorithm. Recursive decomposition can lead to dynamic task generation. For example, consider the recursive decomposition in quicksort. The tasks are generated dynamically, and the size and shape of the task tree is determined by the values in the input array to be sorted. An array of the same size can lead to task-dependency graphs of different shapes and with a different total number of tasks.

Exploratory decomposition can be formulated to generate tasks either statically or dynamically. For example, consider the 15-puzzle problem. One way to generate a static task-dependency graph using exploratory decomposition is as follows. First, a preprocessing task starts with the initial configuration and expands the search tree in a breadth-first manner until a predefined number of configurations are generated. These configuration now represent independent tasks, which can be mapped onto different processes and run independently. A different decomposition that generates tasks dynamically would be one in which a task takes a state as input, expands it through a predefined



number of steps of breadth-first search and spawns new tasks to perform the same computation on each of the resulting states (unless it has found the solution, in which case the algorithm terminates).

**Task Sizes:** The size of a task is the relative amount of time required to complete it. The complexity of mapping schemes often depends on whether or not the tasks are uniform; i.e., whether or not they require roughly the same amount of time. If the amount of time required by the tasks varies significantly, then they are said to be non-uniform.

**Knowledge of Task Sizes:** The third characteristic that influences the choice of mapping scheme is knowledge of the task size. If the size of all the tasks is known, then this information can often be used in mapping of tasks to processes. For example, in the various decompositions for matrix multiplication discussed so far, the computation time for each task is known before the parallel program starts. On the other hand, the size of a typical task in the 15-puzzle problem is unknown. We do not know a priori how many moves will lead to the solution from a given state.


**Size of Data Associated with Tasks** Another important characteristic of a task is the size of data associated with it. The reason this is an important consideration for mapping is that the data associated with a task must be available to the process performing that task, and the size and the location of these data may determine the process that can perform the task without incurring excessive data-movement overheads.

Different types of data associated with a task may have different sizes. For instance, the input data may be small but the output may be large, or vice versa. For example, the input to a task in the 15-puzzle problem may be just one state of the puzzle. This is a small input relative to the amount of computation that may be required to find a sequence of moves from this state to a solution state. In the problem of computing the minimum of a sequence, the size of the input is proportional to the amount of computation, but the output is just one number. In the parallel formulation of the quick sort, the size of both the input and the output data is of the same order as the sequential time needed to solve the task.

### Characteristics of Inter-Task Interactions

In any nontrivial parallel algorithm, tasks need to interact with each other to share data, work, or synchronization information. Different parallel algorithms require different types of interactions among concurrent tasks. The nature of these interactions makes them more suitable for certain programming paradigms and mapping schemes, and less suitable for others. The types of inter-task interactions can be described along different dimensions, each corresponding to a distinct characteristic of the underlying computations.

**Static versus Dynamic** One way of classifying the type of interactions that take place among concurrent tasks is to consider whether or not these interactions have a static or dynamic pattern. An interaction pattern is static if for each task, the interactions happen at predetermined times, and if the set of tasks to interact with at these times is known prior to the execution of the algorithm. In other words, in a static interaction pattern, not only is the task-interaction graph known a priori, but the stage of the computation at which each interaction occurs is also known. An interaction pattern is dynamic if the timing of interactions or the set of tasks to interact with cannot be determined prior to the execution of the algorithm.



Static interactions can be programmed easily in the message-passing paradigm, but dynamic interactions are harder to program. The reason is that interactions in message-passing require active involvement of both interacting tasks – the sender and the receiver of information. The unpredictable nature of dynamic iterations makes it hard for both the sender and the receiver to participate in the interaction at the same time. Therefore, when implementing a parallel algorithm with dynamic interactions in the message-passing paradigm, the tasks must be assigned additional synchronization or polling responsibility. Shared-address space programming can code both types of interactions equally easily.

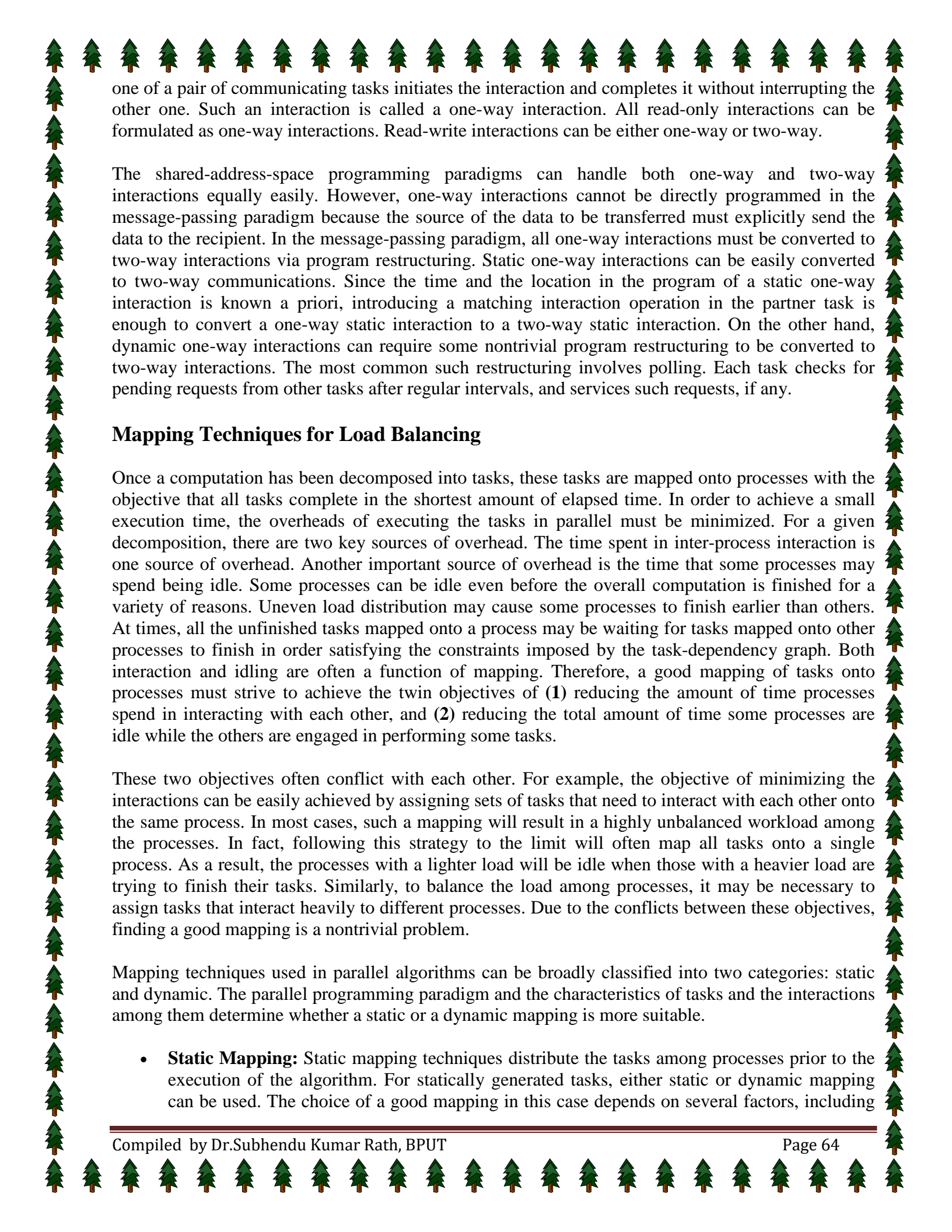
The decompositions for parallel matrix multiplication presented earlier in this chapter exhibit static inter-task interactions. For an example of dynamic interactions, consider solving the 15-puzzle problem in which tasks are assigned different states to explore after an initial step that generates the desirable number of states by applying breadth-first search on the initial state. It is possible that a certain state leads to all dead ends and a task exhausts its search space without reaching the goal state, while other tasks are still busy trying to find a solution. The task that has exhausted its work can pick up an unexplored state from the queue of another busy task and start exploring it. The interactions involved in such a transfer of work from one task to another are dynamic.

**Regular versus Irregular:** Another way of classifying the interactions is based upon their spatial structure. An interaction pattern is considered to be regular if it has some structure that can be exploited for efficient implementation. On the other hand, an interaction pattern is called irregular if no such regular pattern exists. Irregular and dynamic communications are harder to handle, particularly in the message-passing programming paradigm. An example of a decomposition with a regular interaction pattern is the problem of image dithering.

**Read-only versus Read-Write:** We have already learned that sharing of data among tasks leads to inter-task interaction. However, the type of sharing may impact the choice of the mapping. Data sharing interactions can be categorized as either read-only or read-write interactions. As the name suggests, in read-only interactions, tasks require only a read-access to the data shared among many concurrent tasks. For example, in the decomposition for parallel matrix multiplication, the tasks only need to read the shared input matrices A and B. In read-write interactions, multiple tasks need to read and write on some shared data. For example, consider the problem of solving the 15-puzzle. In this formulation, each state is considered an equally suitable candidate for further expansion. The search can be made more efficient if the states that appear to be closer to the solution are given a priority for further exploration. An alternative search technique known as heuristic search implements such a strategy. In heuristic search, we use a heuristic to provide a relative approximate indication of the distance of each state from the solution (i.e. the potential number of moves required to reach the solution). In the case of the 15-puzzle, the number of tiles that are out of place in a given state could serve as such a heuristic. The states that need to be expanded further are stored in a priority queue based on the value of this heuristic. While choosing the states to expand, we give preference to more promising states, i.e. the ones that have fewer out-of-place tiles and hence, are more likely to lead to a quick solution. In this situation, the priority queue constitutes shared data and tasks need both read and write access to it; they need to put the states resulting from an expansion into the queue and they need to pick up the next most promising state for the next expansion.

**One-way versus Two-way:** In some interactions, the data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks. Such interactions are called two-way interactions and usually involve predefined producer and consumer tasks. In other interactions, only





one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one. Such an interaction is called a one-way interaction. All read-only interactions can be formulated as one-way interactions. Read-write interactions can be either one-way or two-way.

The shared-address-space programming paradigms can handle both one-way and two-way interactions equally easily. However, one-way interactions cannot be directly programmed in the message-passing paradigm because the source of the data to be transferred must explicitly send the data to the recipient. In the message-passing paradigm, all one-way interactions must be converted to two-way interactions via program restructuring. Static one-way interactions can be easily converted to two-way communications. Since the time and the location in the program of a static one-way interaction is known a priori, introducing a matching interaction operation in the partner task is enough to convert a one-way static interaction to a two-way static interaction. On the other hand, dynamic one-way interactions can require some nontrivial program restructuring to be converted to two-way interactions. The most common such restructuring involves polling. Each task checks for pending requests from other tasks after regular intervals, and services such requests, if any.


## Mapping Techniques for Load Balancing

Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the objective that all tasks complete in the shortest amount of elapsed time. In order to achieve a small execution time, the overheads of executing the tasks in parallel must be minimized. For a given decomposition, there are two key sources of overhead. The time spent in inter-process interaction is one source of overhead. Another important source of overhead is the time that some processes may spend being idle. Some processes can be idle even before the overall computation is finished for a variety of reasons. Uneven load distribution may cause some processes to finish earlier than others. At times, all the unfinished tasks mapped onto a process may be waiting for tasks mapped onto other processes to finish in order satisfying the constraints imposed by the task-dependency graph. Both interaction and idling are often a function of mapping. Therefore, a good mapping of tasks onto processes must strive to achieve the twin objectives of (1) reducing the amount of time processes spend in interacting with each other, and (2) reducing the total amount of time some processes are idle while the others are engaged in performing some tasks.

These two objectives often conflict with each other. For example, the objective of minimizing the interactions can be easily achieved by assigning sets of tasks that need to interact with each other onto the same process. In most cases, such a mapping will result in a highly unbalanced workload among the processes. In fact, following this strategy to the limit will often map all tasks onto a single process. As a result, the processes with a lighter load will be idle when those with a heavier load are trying to finish their tasks. Similarly, to balance the load among processes, it may be necessary to assign tasks that interact heavily to different processes. Due to the conflicts between these objectives, finding a good mapping is a nontrivial problem.

Mapping techniques used in parallel algorithms can be broadly classified into two categories: static and dynamic. The parallel programming paradigm and the characteristics of tasks and the interactions among them determine whether a static or a dynamic mapping is more suitable.

- **Static Mapping:** Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm. For statically generated tasks, either static or dynamic mapping can be used. The choice of a good mapping in this case depends on several factors, including



the knowledge of task sizes, the size of data associated with tasks, the characteristics of inter-task interactions, and even the parallel programming paradigm. Even when task sizes are known, in general, the problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks. However, for many practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static mapping problem.

Algorithms that make use of static mapping are in general easier to design and program.

- **Dynamic Mapping:** Dynamic mapping techniques distribute the work among processes during the execution of the algorithm. If tasks are generated dynamically, then they must be mapped dynamically too. If task sizes are unknown, then a static mapping can potentially lead to serious load-imbalances and dynamic mappings are usually more effective. If the amount of data associated with tasks is large relative to the computation, then a dynamic mapping may entail moving this data among processes. The cost of this data movement may outweigh some other advantages of dynamic mapping and may render a static mapping more suitable. However, in a shared-address-space paradigm, dynamic mapping may work well even with large data associated with tasks if the interaction is read-only. The reader should be aware that the shared-address-space programming paradigm does not automatically provide immunity against data-movement costs.

Algorithms that require dynamic mapping are usually more complicated, particularly in the message-passing programming paradigm.

### Schemes for Static Mapping

Static mapping is often, though not exclusively, used in conjunction with a decomposition based on data partitioning. Static mapping is also used for mapping certain problems that are expressed naturally by a static task-dependency graph. In the following subsections, we will discuss mapping schemes based on data partitioning and task partitioning.

#### Mappings Based on Data Partitioning

We will discuss mappings based on partitioning two of the most common ways of representing data in algorithms, namely, arrays and graphs. The data-partitioning actually induces a decomposition, but the partitioning or the decomposition is selected with the final mapping in mind.

**Array Distribution Schemes** In a decomposition based on partitioning data, the tasks are closely associated with portions of data by the owner-computes rule. Therefore, mapping the relevant data onto the processes is equivalent to mapping tasks onto processes. We now study some commonly used techniques of distributing arrays or matrices among processes.

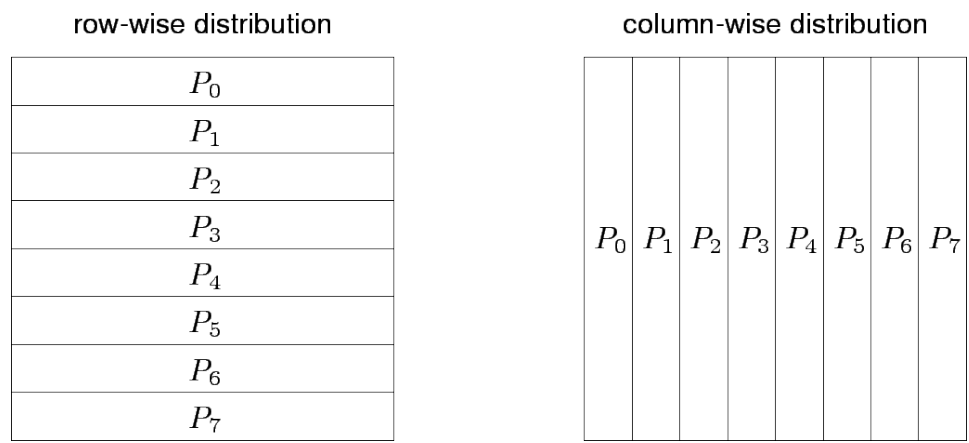
#### Block Distributions

Block distributions are some of the simplest ways to distribute an array and assign uniform contiguous portions of the array to different processes. In these distributions, a d-dimensional array is distributed among the processes such that each process receives a contiguous block of array entries along a specified subset of array dimensions. Block distributions of arrays are particularly suitable

when there is a locality of interaction, i.e., computation of an element of an array requires other nearby elements in the array.

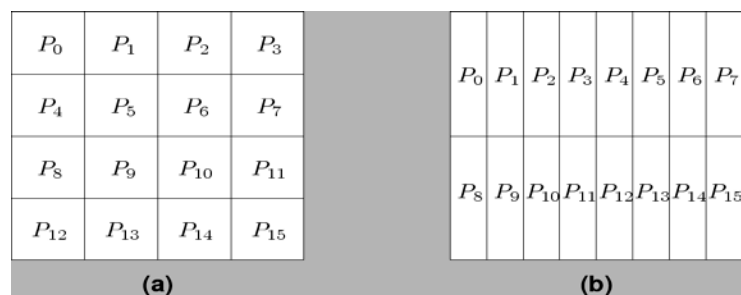
For example, consider an  $n \times n$  two-dimensional array  $A$  with  $n$  rows and  $n$  columns. We can now select one of these dimensions, e.g., the first dimension, and partition the array into  $p$  parts such that the  $k$ th part contains rows  $kn/p \dots (k + 1)n/p - 1$ , where  $0 < k < p$ . That is, each partition contains a block of  $n/p$  consecutive rows of  $A$ . Similarly, if we partition  $A$  along the second dimension, then each partition contains a block of  $n/p$  consecutive columns.

Figure . Examples of one-dimensional partitioning of an array among eight processes.



Similarly, instead of selecting a single dimension, we can select multiple dimensions to partition. For instance, in the case of array  $A$  we can select both dimensions and partition the matrix into blocks such that each block corresponds to a  $n/p_1 \times n/p_2$  section of the matrix, with  $p = p_1 \times p_2$  being the number of processes. The figure illustrates two different two-dimensional distributions, on a  $4 \times 4$  and  $2 \times 8$  process grid, respectively. In general, given a  $d$ -dimensional array, we can distribute it using up to a  $d$ -dimensional block distribution.

Figure . Examples of two-dimensional distributions of an array, (a) on a  $4 \times 4$  process grid, and (b) on a  $2 \times 8$  process grid.

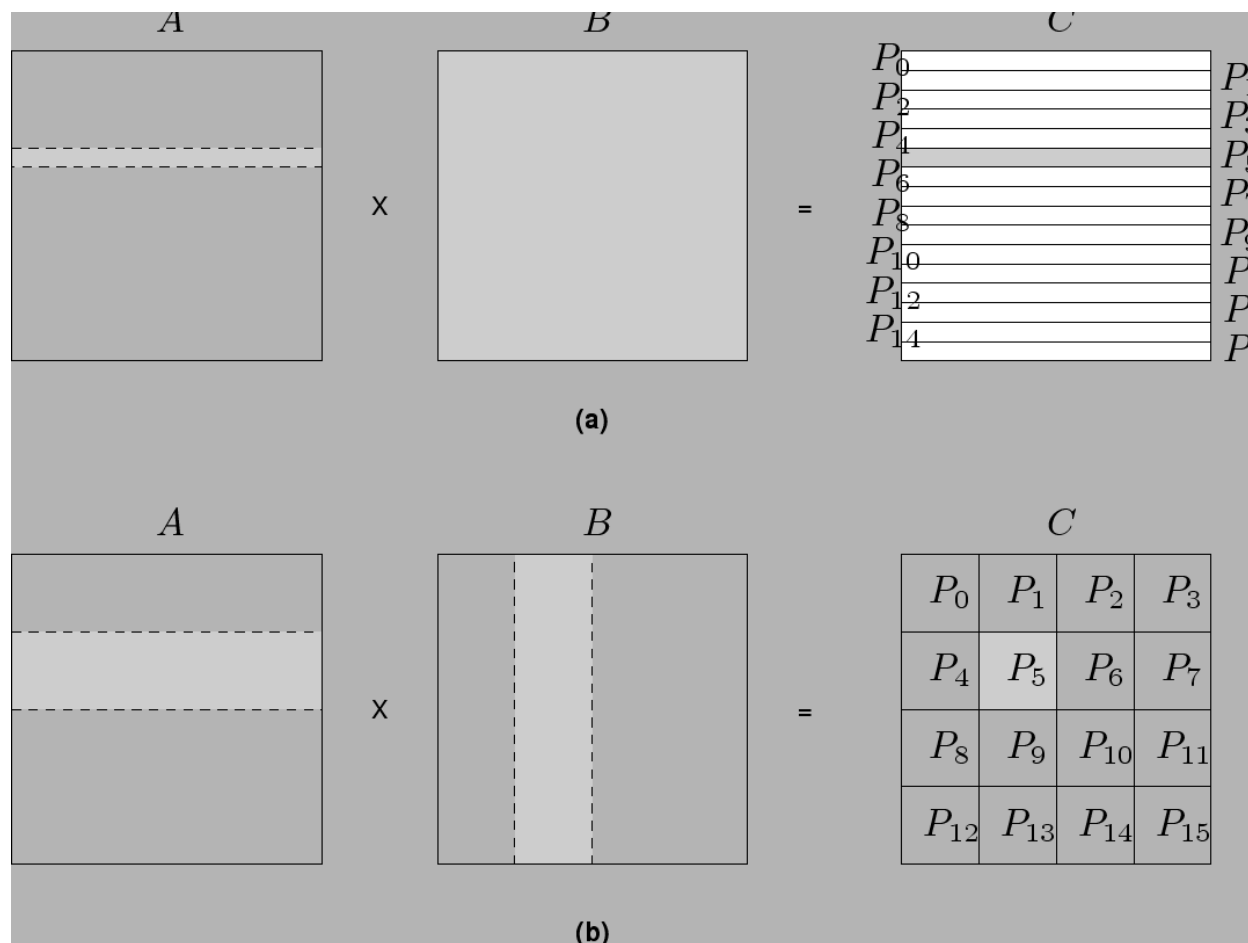


Using these block distributions we can load-balance a variety of parallel computations that operate on multi-dimensional arrays. For example, consider the  $n \times n$  matrix multiplication  $C = A \times B$ . One way of decomposing this computation is to partition the output matrix  $C$ . Since each entry of  $C$  requires the same amount of computation, we can balance the computations by using either a one- or two-dimensional block distribution to partition  $C$  uniformly among the  $p$  available processes. In the first case, each process will get a block of  $n/p$  rows (or columns) of  $C$ , whereas in the second case, each

process will get a block of size. In either case, the process will be responsible for computing the entries of the partition of  $C$  assigned to it.

As the matrix-multiplication example illustrates, quite often we have the choice of mapping the computations using either a one- or a two-dimensional distribution (and even more choices in the case of higher dimensional arrays). In general, higher dimensional distributions allow us to use more processes. For example, in the case of matrix-matrix multiplication, a one-dimensional distribution will allow us to use up to  $n$  processes by assigning a single row of  $C$  to each process. On the other hand, a two-dimensional distribution will allow us to use up to  $n^2$  processes by assigning a single element of  $C$  to each process.

Figure : Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices  $A$  and  $B$  are required by the process that computes the shaded portion of the output matrix  $C$ .



### Cyclic and Block-Cyclic Distributions

If the amount of work differs for different elements of a matrix, a block distribution can potentially lead to load imbalances. A classic example of this phenomenon is LU factorization of a matrix, in which the amount of computation increases from the top left to the bottom right of the matrix.

Example :Dense LU factorization

In its simplest form, the LU factorization algorithm factors a nonsingular square matrix A into the product of a lower triangular matrix L with a unit diagonal and an upper triangular matrix U. The algorithm shows the serial algorithm. Let A be an n x n matrix with rows and columns numbered from 1 to n. The factorization process consists of n major steps – each consisting of an iteration of the outer loop starting at Line 3 in the algorithm. In step k, first, the partial column A[k + 1 : n, k] is divided by A[k, k]. Then, the outer product A[k + 1 : n, k] x A[k, k + 1 : n] is subtracted from the (n - k) x (n - k) submatrix A[k + 1 : n, k + 1 : n]. In a practical implementation of LU factorization, separate arrays are not used for L and U and A is modified to store L and U in its lower and upper triangular parts, respectively. The 1's on the principal diagonal of L are implicit and the diagonal entries actually belong to U after factorization.

**Algorithm :** A serial column-based algorithm to factor a nonsingular matrix A into a lower-triangular matrix L and an upper-triangular matrix U. Matrices L and U share space with A. On Line 9, A[i, j] on the left side of the assignment is equivalent to L [i, j] if i > j; otherwise, it is equivalent to U [i, j].

```

1.  procedure COL_LU (A)
2.  begin
3.      for k := 1 to n do
4.          for j := k to n do
5.              A[j, k] := A[j, k]/A[k, k];
6.          endfor;
7.          for j := k + 1 to n do
8.              for i := k + 1 to n do
9.                  A[i, j] := A[i, j] - A[i, k] x A[k, j];
10.             endfor;
11.         endfor;

```

/\*  
After this iteration, column A[k + 1 : n, k] is logically the kth column of L and row A[k, k : n] is logically the kth row of U.  
\*/

```

12.     endfor;
13. end COL_LU

```

**Figure . A decomposition of LU factorization into 14 tasks.**

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

$$A_{1,1} \rightarrow L_{1,1}U_{1,1}$$

$$A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$$

$$L_{3,2} = A_{3,2}U_{2,2}^{-1}$$

$$L_{2,1} = A_{2,1}U_{1,1}^{-1}$$

$$A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$$

$$U_{2,3} = L_{2,2}^{-1}A_{2,3}$$

$$L_{3,1} = A_{3,1}U_{1,1}^{-1}$$

$$A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$$

$$A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$$

$$U_{1,2} = L_{1,1}^{-1}A_{1,2}$$

$$A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$$

$$A_{3,3} \rightarrow L_{3,3}U_{3,3}$$

$$U_{1,3} = L_{1,1}^{-1}A_{1,3}$$

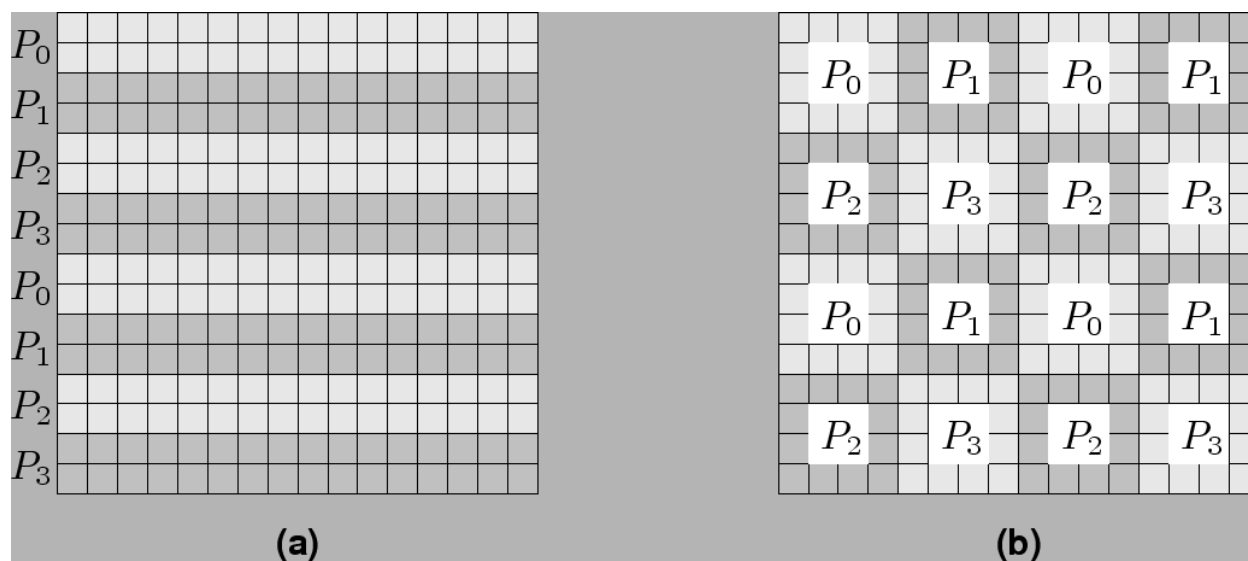
$$A_{2,2} \rightarrow L_{2,2}U_{2,2}$$

## Randomized Block Distributions

A block-cyclic distribution may not always be able to balance computations when the distribution of work has some special patterns.

Randomized block distribution, a more general form of the block distribution. Just like a block-cyclic distribution, load balance is sought by partitioning the array into many more blocks than the number of available processes. However, the blocks are uniformly and randomly distributed among the processes.

Figure : Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

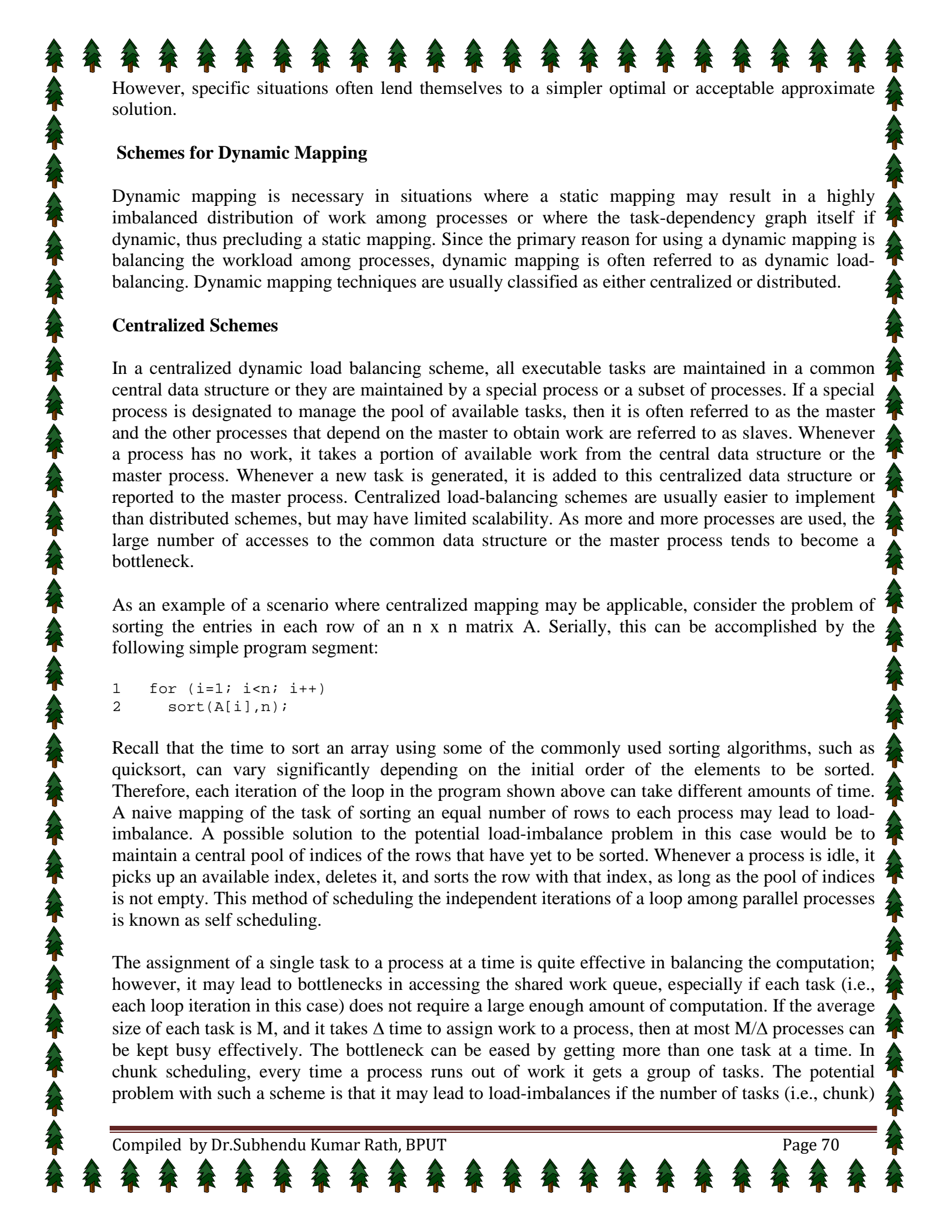


## Graph Partitioning

The array-based distribution schemes that we described so far are quite effective in balancing the computations and minimizing the interactions for a wide range of algorithms that use dense matrices and have structured and regular interaction patterns. However, there are many algorithms that operate on sparse data structures and for which the pattern of interaction among data elements is data dependent and highly irregular. Numerical simulations of physical phenomena provide a large source of such type of computations. In these computations, the physical domain is discretized and represented by a mesh of elements. The simulation of the physical phenomenon being modeled then involves computing the values of certain physical quantities at each mesh point. The computation at a mesh point usually requires data corresponding to that mesh point and to the points that are adjacent to it in the mesh.

## Mappings Based on Task Partitioning

A mapping based on partitioning a task-dependency graph and mapping its nodes onto processes can be used when the computation is naturally expressible in the form of a static task-dependency graph with tasks of known sizes. As usual, this mapping must seek to achieve the often conflicting objectives of minimizing idle time and minimizing the interaction time of the parallel algorithm. Determining an optimal mapping for a general task-dependency graph is an NP-complete problem.



However, specific situations often lend themselves to a simpler optimal or acceptable approximate solution.

## Schemes for Dynamic Mapping

Dynamic mapping is necessary in situations where a static mapping may result in a highly imbalanced distribution of work among processes or where the task-dependency graph itself is dynamic, thus precluding a static mapping. Since the primary reason for using a dynamic mapping is balancing the workload among processes, dynamic mapping is often referred to as dynamic load-balancing. Dynamic mapping techniques are usually classified as either centralized or distributed.

### Centralized Schemes


In a centralized dynamic load balancing scheme, all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes. If a special process is designated to manage the pool of available tasks, then it is often referred to as the master and the other processes that depend on the master to obtain work are referred to as slaves. Whenever a process has no work, it takes a portion of available work from the central data structure or the master process. Whenever a new task is generated, it is added to this centralized data structure or reported to the master process. Centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability. As more and more processes are used, the large number of accesses to the common data structure or the master process tends to become a bottleneck.

As an example of a scenario where centralized mapping may be applicable, consider the problem of sorting the entries in each row of an  $n \times n$  matrix  $A$ . Serially, this can be accomplished by the following simple program segment:

```
1  for (i=1; i<n; i++)
2      sort(A[i],n);
```

Recall that the time to sort an array using some of the commonly used sorting algorithms, such as quicksort, can vary significantly depending on the initial order of the elements to be sorted. Therefore, each iteration of the loop in the program shown above can take different amounts of time. A naive mapping of the task of sorting an equal number of rows to each process may lead to load-imbalance. A possible solution to the potential load-imbalance problem in this case would be to maintain a central pool of indices of the rows that have yet to be sorted. Whenever a process is idle, it picks up an available index, deletes it, and sorts the row with that index, as long as the pool of indices is not empty. This method of scheduling the independent iterations of a loop among parallel processes is known as self scheduling.

The assignment of a single task to a process at a time is quite effective in balancing the computation; however, it may lead to bottlenecks in accessing the shared work queue, especially if each task (i.e., each loop iteration in this case) does not require a large enough amount of computation. If the average size of each task is  $M$ , and it takes  $\Delta$  time to assign work to a process, then at most  $M/\Delta$  processes can be kept busy effectively. The bottleneck can be eased by getting more than one task at a time. In chunk scheduling, every time a process runs out of work it gets a group of tasks. The potential problem with such a scheme is that it may lead to load-imbalances if the number of tasks (i.e., chunk)



assigned in a single step is large. The danger of load-imbalance due to large chunk sizes can be reduced by decreasing the chunk-size as the program progresses. That is, initially the chunk size is large, but as the number of iterations left to be executed decreases, the chunk size also decreases. A variety of schemes have been developed for gradually adjusting the chunk size, that decrease the chunk size either linearly or non-linearly.

### **Distributed Schemes**

In a distributed dynamic load balancing scheme, the set of executable tasks are distributed among processes which exchange tasks at run time to balance work. Each process can send work to or receive work from any other process. These methods do not suffer from the bottleneck associated with the centralized schemes. Some of the critical parameters of a distributed load balancing scheme are as follows:

- How are the sending and receiving processes paired together?
- Is the work transfer initiated by the sender or the receiver?
- How much work is transferred in each exchange? If too little work is transferred, then the receiver may not receive enough work and frequent transfers resulting in excessive interaction may be required. If too much work is transferred, then the sender may be out of work soon, again resulting in frequent transfers.
- When is the work transfer performed? For example, in receiver initiated load balancing, work may be requested when the process has actually run out of work or when the receiver has too little work left and anticipates being out of work soon.

### **Suitability to Parallel Architectures**

Note that, in principle, both centralized and distributed mapping schemes can be implemented in both message-passing and shared-address-space paradigms. However, by its very nature any dynamic load balancing scheme requires movement of tasks from one process to another. Hence, for such schemes to be effective on message-passing computers, the size of the tasks in terms of computation should be much higher than the size of the data associated with the tasks. In a shared-address-space paradigm, the tasks don't need to be moved explicitly, although there is some implied movement of data to local caches or memory banks of processes. In general, the computational granularity of tasks to be moved can be much smaller on shared-address architecture than on message-passing architectures.


### **Methods for Containing Interaction Overheads**

As noted earlier, reducing the interaction overhead among concurrent tasks is important for an efficient parallel program. The overhead that a parallel program incurs due to interaction among its processes depends on many factors, such as the volume of data exchanged during interactions, the frequency of interaction, the spatial and temporal pattern of interactions, etc.

#### **Maximizing Data Locality**

In most nontrivial parallel programs, the tasks executed by different processes require access to some common data. For example, in sparse matrix-vector multiplication  $y = Ab$ , in which tasks correspond to computing individual elements of vector  $y$ , all elements of the input vector  $b$  need to be accessed by multiple tasks. In addition to sharing the original input data, interaction may result if processes





require data generated by other processes. The interaction overheads can be reduced by using techniques that promote the use of local data or data that have been recently fetched. Data locality enhancing techniques encompass a wide range of schemes that try to minimize the volume of nonlocal data that are accessed, maximize the reuse of recently accessed data, and minimize the frequency of accesses. In many cases, these schemes are similar in nature to the data reuse optimizations often performed in modern cache based microprocessors.


### **Minimize Volume of Data-Exchange**

A fundamental technique for reducing the interaction overhead is to minimize the overall volume of shared data that needs to be accessed by concurrent processes. This is akin to maximizing the temporal data locality, i.e., making as many of the consecutive references to the same data as possible. Clearly, performing as much of the computation as possible using locally available data obviates the need for bringing in more data into local memory or cache for a process to perform its tasks. As discussed previously, one way of achieving this is by using appropriate decomposition and mapping schemes.

Another way of decreasing the amount of shared data that are accessed by multiple processes is to use local data to store intermediate results, and perform the shared data access to only place the final results of the computation. For example, consider computing the dot product of two vectors of length  $n$  in parallel such that each of the  $p$  tasks multiplies  $n/p$  pairs of elements. Rather than adding each individual product of a pair of numbers to the final result, each task can first create a partial dot product of its assigned portion of the vectors of length  $n/p$  in its own local location, and only access the final shared location once to add this partial result. This will reduce the number of accesses to the shared location where the result is stored to  $p$  from  $n$ .

**Minimize Frequency of Interactions** Minimizing interaction frequency is important in reducing the interaction overheads in parallel programs because there is a relatively high startup cost associated with each interaction on many architectures. Interaction frequency can be reduced by restructuring the algorithm such that shared data are accessed and used in large pieces. Thus, by amortizing the startup cost over large accesses, we can reduce the overall interaction overhead, even if such restructuring does not necessarily reduce the overall volume of shared data that need to be accessed. This is akin to increasing the spatial locality of data access, i.e., ensuring the proximity of consecutively accessed data locations. On a shared-address-space architecture, each time a word is accessed, an entire cache line containing many words is fetched. If the program is structured to have spatial locality, then fewer cache lines are accessed. On a message-passing system, spatial locality leads to fewer message-transfers over the network because each message can transfer larger amounts of useful data. The number of messages can sometimes be reduced further on a message-passing system by combining messages between the same source-destination pair into larger messages if the interaction pattern permits and if the data for multiple messages are available at the same time, albeit in separate data structures.

Sparse matrix-vector multiplication is a problem whose parallel formulation can use this technique to reduce interaction overhead. In typical applications, repeated sparse matrix-vector multiplication is performed with matrices of the same nonzero pattern but different numerical nonzero values. While solving this problem in parallel, a process interacts with others to access elements of the input vector that it may need for its local computation. Through a one-time scanning of the nonzero pattern of the rows of the sparse matrix that a process is responsible for, it can determine exactly which elements of



the input vector it needs and from which processes. Then, before starting each multiplication, a process can first collect all the nonlocal entries of the input vector that it requires, and then perform an interaction-free multiplication. This strategy is far superior than trying to access a nonlocal element of the input vector as and when required in the computation.

### **Minimizing Contention and Hot Spots**

Our discussion so far has been largely focused on reducing interaction overheads by directly or indirectly reducing the frequency and volume of data transfers. However, the data-access and inter-task interaction patterns can often lead to contention that can increase the overall interaction overhead. In general, contention occurs when multiple tasks try to access the same resources concurrently. Multiple simultaneous transmissions of data over the same interconnection link, multiple simultaneous accesses to the same memory block, or multiple processes sending messages to the same process at the same time, can all lead to contention. This is because only one of the multiple operations can proceed at a time and the others are queued and proceed sequentially.


### **Overlapping Computations with Interactions**

The amount of time that processes spend waiting for shared data to arrive or to receive additional work after an interaction has been initiated can be reduced, often substantially, by doing some useful computations during this waiting time. There are a number of techniques that can be used to overlap computations with interactions.

A simple way of overlapping is to initiate an interaction early enough so that it is completed before it is needed for computation. To achieve this, we must be able to identify computations that can be performed before the interaction and do not depend on it. Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original algorithm. Typically, this is possible if the interaction pattern is spatially and temporally static (and therefore, predictable) or if multiple tasks that are ready for execution are available on the same process so that if one blocks to wait for an interaction to complete, the process can work on another task. The reader should note that by increasing the number of parallel tasks to promote computation-interaction overlap, we are reducing the granularity of the tasks, which in general tends to increase overheads. Therefore, this technique must be used judiciously.

In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process. It then waits for the request to be serviced. If the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance, then it may continue towards finishing the tasks at hand while the request for more work is being serviced. Depending on the problem, estimating the amount of remaining work may be easy or hard.

In most cases, overlapping computations with interaction requires support from the programming paradigm, the operating system, and the hardware. The programming paradigm must provide a mechanism to allow interactions and computations to proceed concurrently. This mechanism should be supported by the underlying hardware. Disjoint address-space paradigms and architectures usually provide this support via non-blocking message passing primitives. The programming paradigm provides functions for sending and receiving messages that return control to the user's program before they have actually completed. Thus, the program can use these primitives to initiate the interactions,



and then proceed with the computations. If the hardware permits computation to proceed concurrently with message transfers, then the interaction overhead can be reduced significantly.

On a shared-address-space architecture, the overlapping of computations and interaction is often assisted by prefetching hardware. In this case, an access to shared data is nothing more than a regular load or store instruction. The prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed. In the absence of prefetching hardware, the same effect can be achieved by a compiler that detects the access pattern and places pseudo-references to certain key memory locations before these locations are actually utilized by the computation. The degree of success of this scheme is dependent upon the available structure in the program that can be inferred by the prefetch hardware and by the degree of independence with which the prefetch hardware can function while computation is in progress.

### **Replicating Data or Computations**

Replication of data or computations is another technique that may be useful in reducing interaction overheads.


In some parallel algorithms, multiple processes may require frequent read-only access to shared data structure, such as a hash-table, in an irregular pattern. Unless the additional memory requirements are prohibitive, it may be best in a situation like this to replicate a copy of the shared data structure on each process so that after the initial interaction during replication, all subsequent accesses to this data structure are free of any interaction overhead.

In the shared-address-space paradigm, replication of frequently accessed read-only data is often affected by the caches without explicit programmer intervention. Explicit data replication is particularly suited for architectures and programming paradigms in which read-only access to shared data is significantly more expensive or harder to express than local data accesses. Therefore, the message-passing programming paradigm benefits the most from data replication, which may reduce interaction overhead and also significantly simplify the writing of the parallel program.

Data replication, however, does not come without its own cost. Data replication increases the memory requirements of a parallel program. The aggregate amount of memory required to store the replicated data increases linearly with the number of concurrent processes. This may limit the size of the problem that can be solved on a given parallel computer. For this reason, data replication must be used selectively to replicate relatively small amounts of data.

### **Using Optimized Collective Interaction Operations**

As discussed, often the interaction patterns among concurrent activities are static and regular. A class of such static and regular interaction patterns are those that are performed by groups of tasks, and they are used to achieve regular data accesses or to perform certain type of computations on distributed data. A number of key such collective interaction operations have been identified that appear frequently in many parallel algorithms. Broadcasting some data to all the processes or adding up numbers, each belonging to a different process, are examples of such collective operations. The collective data-sharing operations can be classified into three categories. The first category contains operations that are used by the tasks to access data, the second category of operations are used to



perform some communication-intensive computations, and finally, the third category is used for synchronization.

## Overlapping Interactions with Other Interactions

If the data-transfer capacity of the underlying hardware permits, then overlapping interactions between multiple pairs of processes can reduce the effective volume of communication. As an example of overlapping interactions, consider the commonly used collective communication operation of one-to-all broadcast in a message-passing paradigm with four processes  $P_0$ ,  $P_1$ ,  $P_2$ , and  $P_3$ . A commonly used algorithm to broadcast some data from  $P_0$  to all other processes works as follows. In the first step,  $P_0$  sends the data to  $P_2$ . In the second step,  $P_0$  sends the data to  $P_1$ , and concurrently,  $P_2$  sends the same data that it had received from  $P_0$  to  $P_3$ . The entire operation is thus complete in two steps because the two interactions of the second step require only one time step.

## 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.

### 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. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called data parallelism. The work may be done in phases and the data operated upon in different phases may be different. Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks. Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.

Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms. However, the partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality. On the other hand, shared-address space can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.

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.

### 2 The Task Graph Model



In the task graph model, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs. This model is typically employed to solve problems in which the amount of data associated with the tasks is large relative to the amount of computation associated with them. Usually, tasks are mapped statically to help optimize the cost of data movement among tasks. Sometimes a decentralized dynamic mapping may be used, but even then, the mapping uses the information about the task-dependency graph structure and the interaction pattern of tasks to minimize interaction overhead. Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.

Typical 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, 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.

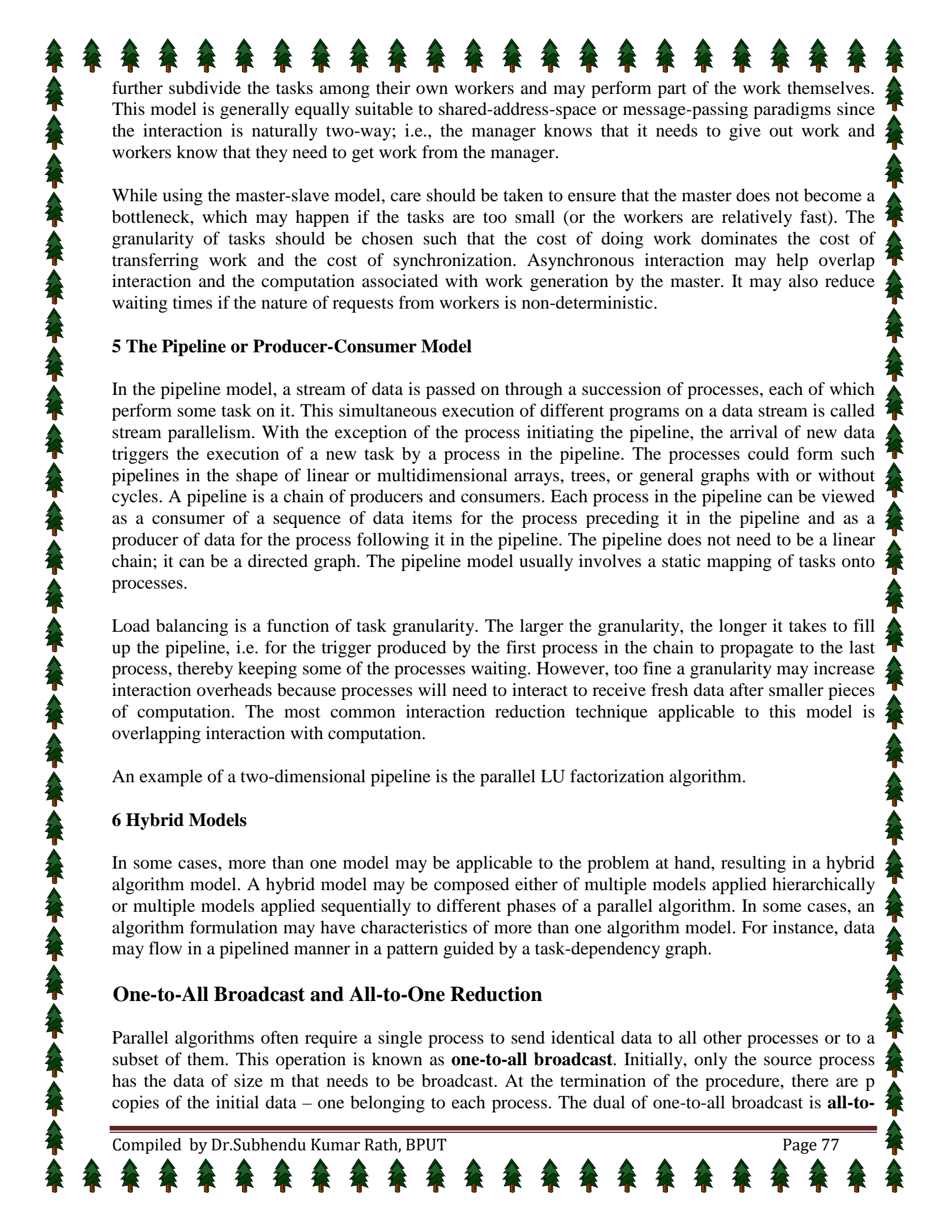
### 3 The Work Pool Model

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. The mapping may be centralized or decentralized. Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure. The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool.

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.

### 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. Usually, there is no desired premapping of work to processes, and any worker can do any job assigned to it. The manager-worker model can be generalized to the hierarchical or multi-level manager-worker model in which the top-level manager feeds large chunks of tasks to second-level managers, who



further subdivide the tasks among their own workers and may perform part of the work themselves. This model is generally equally suitable to shared-address-space or message-passing paradigms since the interaction is naturally two-way; i.e., the manager knows that it needs to give out work and workers know that they need to get work from the manager.

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.

## 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. The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles. A pipeline is a chain of producers and consumers. Each process in the pipeline can be viewed as a consumer of a sequence of data items for the process preceding it in the pipeline and as a producer of data for the process following it in the pipeline. The pipeline does not need to be a linear chain; it can be a directed graph. The pipeline model usually involves a static mapping of tasks onto processes.

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.

An example of a two-dimensional pipeline is the parallel LU factorization algorithm.

## 6 Hybrid Models

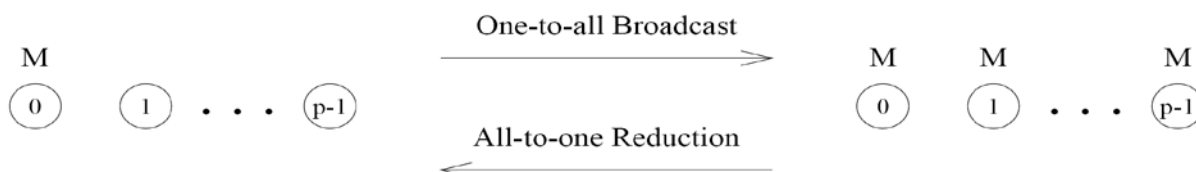
In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. In some cases, an algorithm 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.

## 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  $i$ th word of the accumulated  $M$  is the sum, product, maximum, or minimum of the  $i$ th words of each of the original buffers. The figure shows one-to-all broadcast and all-to-one reduction among  $p$  processes.

Figure 1. One-to-all broadcast and all-to-one reduction.



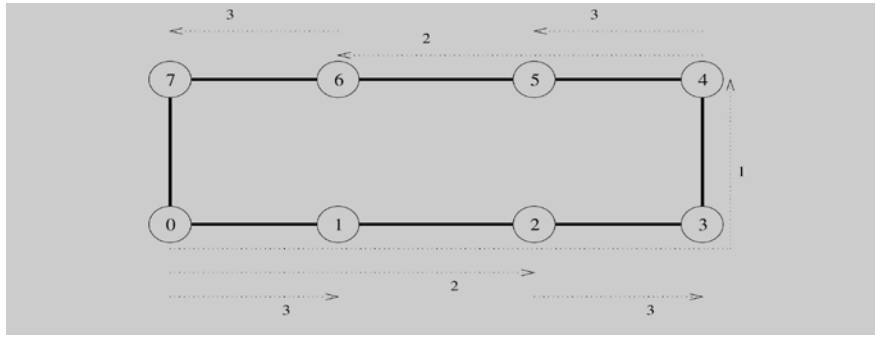
One-to-all broadcast and all-to-one reduction are used in several important parallel algorithms including matrix-vector multiplication, Gaussian elimination, shortest paths, and vector inner product. In the following subsections, we consider the implementation of one-to-all broadcast in detail on a variety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

### 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 the figure. 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.

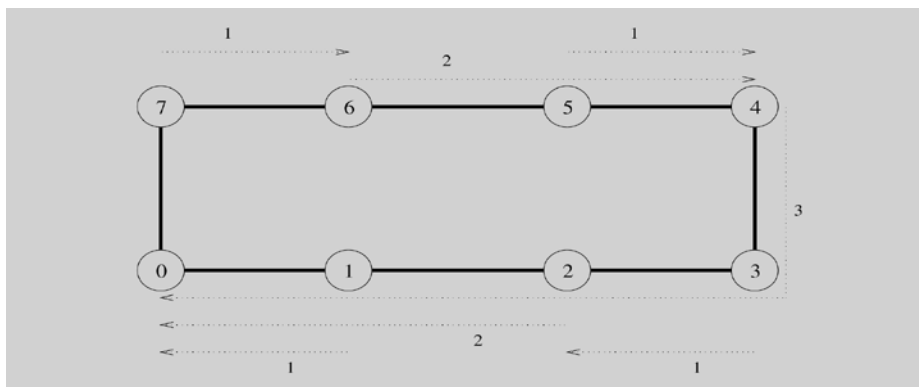
Figure 2. One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.



Note that on a linear array, the destination node to which the message is sent in each step must be carefully chosen. In Figure 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 4.3. 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. Reduction on an eight-node ring with node 0 as the destination of the reduction.

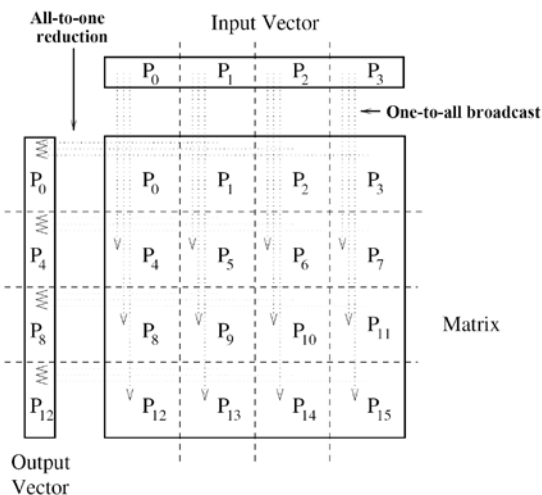


Example : Matrix-vector multiplication

Consider the problem of multiplying an  $n \times n$  matrix  $A$  with an  $n \times 1$  vector  $x$  on an  $n \times n$  mesh of nodes to yield an  $n \times 1$  result vector  $y$ . Algorithm 1 shows a serial algorithm for this problem. Figure 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 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  $n \times n$  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.

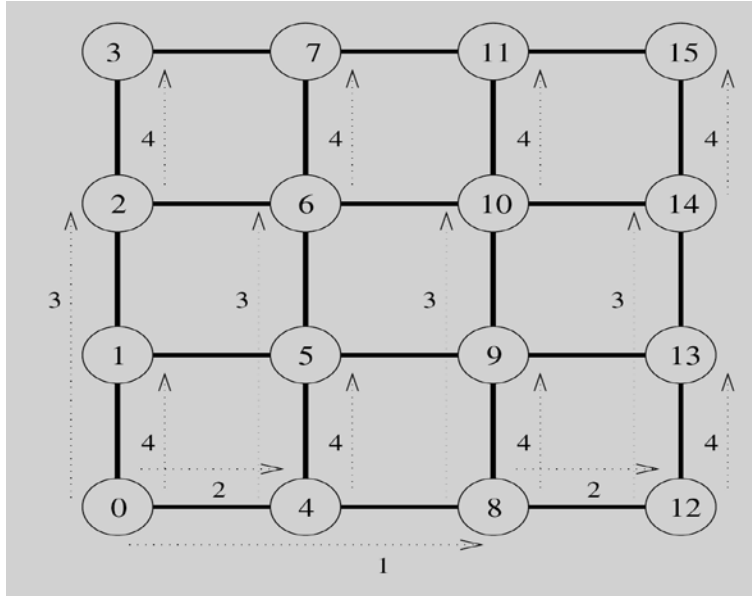
For example,  $P_9$  will receive  $x[1]$  from  $P_1$  as a result of the broadcast, will multiply it with  $A[2, 1]$  and will participate in an all-to-one reduction with  $P_8, P_{10}$ , and  $P_{11}$  to accumulate  $y[2]$  on  $P_8$ .

### Mesh

We can regard each row and column of a square mesh of  $p$  nodes as a linear array of  $\sqrt{p}$  nodes. So a number of communication algorithms on the mesh are simple extensions of their linear array counterparts. A linear array communication operation can be performed in two phases on a mesh. In the first phase, the operation is performed along one or all rows by treating the rows as linear arrays. In the second phase, the columns are treated similarly.

Consider the problem of one-to-all broadcast on a two-dimensional square mesh with  $\sqrt{p}$  rows and  $\sqrt{p}$  columns. First, a one-to-all broadcast is performed from the source to the remaining  $(\sqrt{p} - 1)$  nodes of the same row. Once all the nodes in a row of the mesh have acquired the data, they initiate a one-to-all broadcast in their respective columns. At the end of the second phase, every node in the mesh has a copy of the initial message. The communication steps for one-to-all broadcast on a mesh are illustrated in the figure, for  $p = 16$ , with node 0 at the bottom-left corner as the source. Steps 1 and 2 correspond to the first phase, and steps 3 and 4 correspond to the second phase.

Figure 5. One-to-all broadcast on a 16-node mesh.



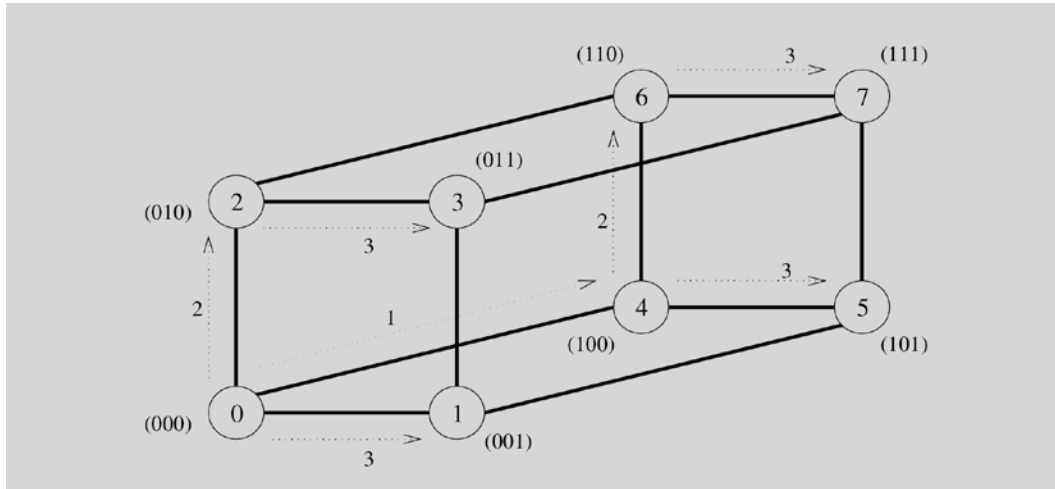
We can use a similar procedure for one-to-all broadcast on a three-dimensional mesh as well. In this case, rows of  $p^{1/3}$  nodes in each of the three dimensions of the mesh would be treated as linear arrays. As in the case of a linear array, reduction can be performed on two- and three-dimensional meshes by simply reversing the direction and the order of messages.

### Hypercube

The previous subsection showed that one-to-all broadcast is performed in two phases on a two-dimensional mesh, with the communication taking place along a different dimension in each phase. Similarly, the process is carried out in three phases on a three-dimensional mesh. A hypercube with  $2^d$  nodes can be regarded as a d-dimensional mesh with two nodes in each dimension. Hence, the mesh algorithm can be extended to the hypercube, except that the process is now carried out in d steps – one in each dimension.

Figure 6 shows a one-to-all broadcast on an eight-node (three-dimensional) hypercube with node 0 as the source. In this figure, communication starts along the highest dimension (that is, the dimension specified by the most significant bit of the binary representation of a node label) and proceeds along successively lower dimensions in subsequent steps. Note that the source and the destination nodes in three communication steps of the algorithm shown in Figure 6 are identical to the ones in the broadcast algorithm on a linear array shown in Figure 2. However, on a hypercube, the order in which the dimensions are chosen for communication does not affect the outcome of the procedure. Figure 6 shows only one such order. Unlike a linear array, the hypercube broadcast would not suffer from congestion if node 0 started out by sending the message to node 1 in the first step, followed by nodes 0 and 1 sending messages to nodes 2 and 3, respectively, and finally nodes 0, 1, 2, and 3 sending messages to nodes 4, 5, 6, and 7, respectively.

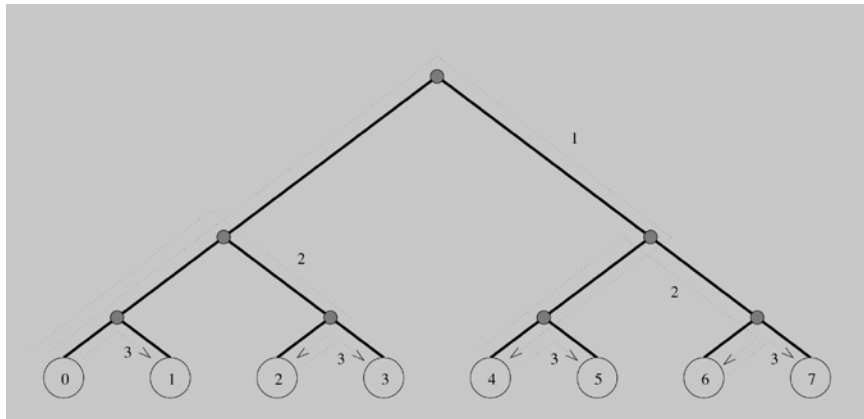
Figure 6. One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses.



### Balanced Binary Tree

The hypercube algorithm for one-to-all broadcast maps naturally onto a balanced binary tree in which each leaf is a processing node and intermediate nodes serve only as switching units. This is illustrated in Figure 7 for eight nodes. In this figure, the communicating nodes have the same labels as in the hypercube algorithm illustrated in Figure 6. Figure 7 shows that there is no congestion on any of the communication links at any time. The difference between the communication on a hypercube and the tree shown in Figure 7 is that there is a different number of switching nodes along different paths on the tree.

Figure . One-to-all broadcast on an eight-node tree.



### Detailed Algorithms

Algorithm 1 shows a one-to-all broadcast procedure on a  $2^d$ -node network when node 0 is the source of the broadcast. The procedure is executed at all the nodes. At any node, the value of `my_id` is the label of that node. Let `X` be the message to be broadcast, which initially resides at the source node 0. The procedure performs `d` communication steps, one along each dimension of a hypothetical hypercube. In Algorithm 1, communication proceeds from the highest to the lowest dimension (although the order in which dimensions are chosen does not matter). The loop counter `i` indicates the current dimension of the hypercube in which communication is taking place. Only the nodes with zero in the `i` least significant bits of their labels participate in communication along dimension `i`. For

instance, on the three-dimensional hypercube shown in Figure 6,  $i$  is equal to 2 in the first time step. Therefore, only nodes 0 and 4 communicate, since their two least significant bits are zero. In the next time step, when  $i = 1$ , all nodes (that is, 0, 2, 4, and 6) with zero in their least significant bits participate in communication. The procedure terminates after communication has taken place along all dimensions.

The variable mask helps determine which nodes communicate in a particular iteration of the loop. The variable mask has  $d$  ( $= \log p$ ) bits, all of which are initially set to one (Line 3). At the beginning of each iteration, the most significant nonzero bit of mask is reset to zero (Line 5). Line 6 determines which nodes communicate in the current iteration of the outer loop. For instance, for the hypercube of Figure 4.6, mask is initially set to 111, and it would be 011 during the iteration corresponding to  $i = 2$  (the  $i$  least significant bits of mask are ones). The AND operation on Line 6 selects only those nodes that have zeros in their  $i$  least significant bits.

Among the nodes selected for communication along dimension  $i$ , the nodes with a zero at bit position  $i$  send the data, and the nodes with a one at bit position  $i$  receive it. The test to determine the sending and receiving nodes is performed on Line 7. For example, in Figure 6, node 0 (000) is the sender and node 4 (100) is the receiver in the iteration corresponding to  $i = 2$ . Similarly, for  $i = 1$ , nodes 0 (000) and 4 (100) are senders while nodes 2 (010) and 6 (110) are receivers.

Algorithm 1 works only if node 0 is the source of the broadcast. For an arbitrary source, we must relabel the nodes of the hypothetical hypercube by XORing the label of each node with the label of the source node before we apply this procedure. A modified one-to-all broadcast procedure that works for any value of source between 0 and  $p - 1$  is shown in Algorithm 2. By performing the XOR operation at Line 3, Algorithm 2 relabels the source node to 0, and relabels the other nodes relative to the source. After this relabeling, the algorithm of Algorithm 1 can be applied to perform the broadcast.

Algorithm 3 gives a procedure to perform an all-to-one reduction on a hypothetical  $d$ -dimensional hypercube such that the final result is accumulated on node 0. Single node-accumulation is the dual of one-to-all broadcast. Therefore, we obtain the communication pattern required to implement reduction by reversing the order and the direction of messages in one-to-all broadcast. Procedure ALL\_TO\_ONE\_REDUCE( $d$ , my\_id,  $m$ ,  $X$ , sum) shown in Algorithm 4.3 is very similar to procedure ONE\_TO\_ALL\_BC( $d$ , my\_id,  $X$ ) shown in Algorithm 4.1. One difference is that the communication in all-to-one reduction proceeds from the lowest to the highest dimension. This change is reflected in the way that variables mask and  $i$  are manipulated in Algorithm 4.3. The criterion for determining the source and the destination among a pair of communicating nodes is also reversed (Line 7). Apart from these differences, procedure ALL\_TO\_ONE\_REDUCE has extra instructions (Lines 13 and 14) to add the contents of the messages received by a node in each iteration (any associative operation can be used in place of addition).

**Algorithm 1 One-to-all broadcast of a message  $X$  from node 0 of a  $d$ -dimensional  $p$ -node hypercube ( $d = \log p$ ). AND and XOR are bitwise logical-and and exclusive-or operations, respectively.**

```

1.  procedure ONE_TO_ALL_BC( $d$ , my_id,  $X$ )
2.  begin
3.      mask :=  $2^d - 1$ ;                /* Set all  $d$  bits of mask to 1 */
4.      for  $i$  :=  $d - 1$  downto 0 do      /* Outer loop */
5.          mask := mask XOR  $2^i$ ;      /* Set bit  $i$  of mask to 0 */

```

```

6.         if (my_id AND mask) = 0 then /* If lower i bits of my_id are 0 */
7.             if (my_id AND 2i) = 0 then
8.                 msg_destination := my_id XOR 2i;
9.                 send X to msg_destination;
10.            else
11.                msg_source := my_id XOR 2i;
12.                receive X from msg_source;
13.            endelse;
14.        endif;
15.    endfor;
16. end ONE_TO_ALL_BC

```

**Algorithm 2 One-to-all broadcast of a message X initiated by source on a d-dimensional hypothetical hypercube. The AND and XOR operations are bitwise logical operations.**

```

1.  procedure GENERAL_ONE_TO_ALL_BC(d, my_id, source, X)
2.  begin
3.      my_virtual_id := my_id XOR source;
4.      mask := 2d - 1;
5.      for i := d - 1 downto 0 do /* Outer loop */
6.          mask := mask XOR 2i; /* Set bit i of mask to 0 */
7.          if (my_virtual_id AND mask) = 0 then
8.              if (my_virtual_id AND 2i) = 0 then
9.                  virtual_dest := my_virtual_id XOR 2i;
10.                 send X to (virtual_dest XOR source);
11.             /* Convert virtual_dest to the label of the physical destination */
12.             else
13.                 virtual_source := my_virtual_id XOR 2i;
14.                 receive X from (virtual_source XOR source);
15.             /* Convert virtual_source to the label of the physical source */
16.             endelse;
17.         endfor;
18. end GENERAL_ONE_TO_ALL_BC

```

**Algorithm 3 Single-node accumulation on a d-dimensional hypercube. Each node contributes a message X containing m words, and node 0 is the destination of the sum. The AND and XOR operations are bitwise logical operations.**

```

1.  procedure ALL_TO_ONE_REDUCE(d, my_id, m, X, sum)
2.  begin
3.      for j := 0 to m - 1 do sum[j] := X[j];
4.      mask := 0;
5.      for i := 0 to d - 1 do
6.          /* Select nodes whose lower i bits are 0 */
7.          if (my_id AND mask) = 0 then
8.              if (my_id AND 2i) ≠ 0 then
9.                  msg_destination := my_id XOR 2i;
10.                 send sum to msg_destination;
11.             else
12.                 msg_source := my_id XOR 2i;
13.                 receive X from msg_source;
14.                 for j := 0 to m - 1 do
15.                     sum[j] := sum[j] + X[j];
16.                 endfor;
17.             endelse;
18.             mask := mask XOR 2i; /* Set bit i of mask to 1 */
19.         endfor;
20. end ALL_TO_ONE_REDUCE

```

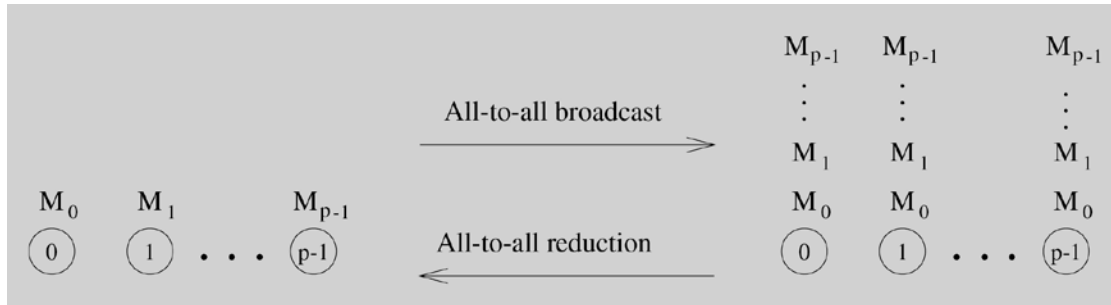
## Cost Analysis

Analyzing the cost of one-to-all broadcast and all-to-one reduction is fairly straightforward. Assume that  $p$  processes participate in the operation and the data to be broadcast or reduced contains  $m$  words. The broadcast or reduction procedure involves  $\log p$  point-to-point simple message transfers, each at a time cost of  $t_s + t_w m$ . Therefore, the total time taken by the procedure is

## 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 (Problem 4.8). Figure 4.8 illustrates all-to-all broadcast and all-to-all reduction.

Figure : 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.

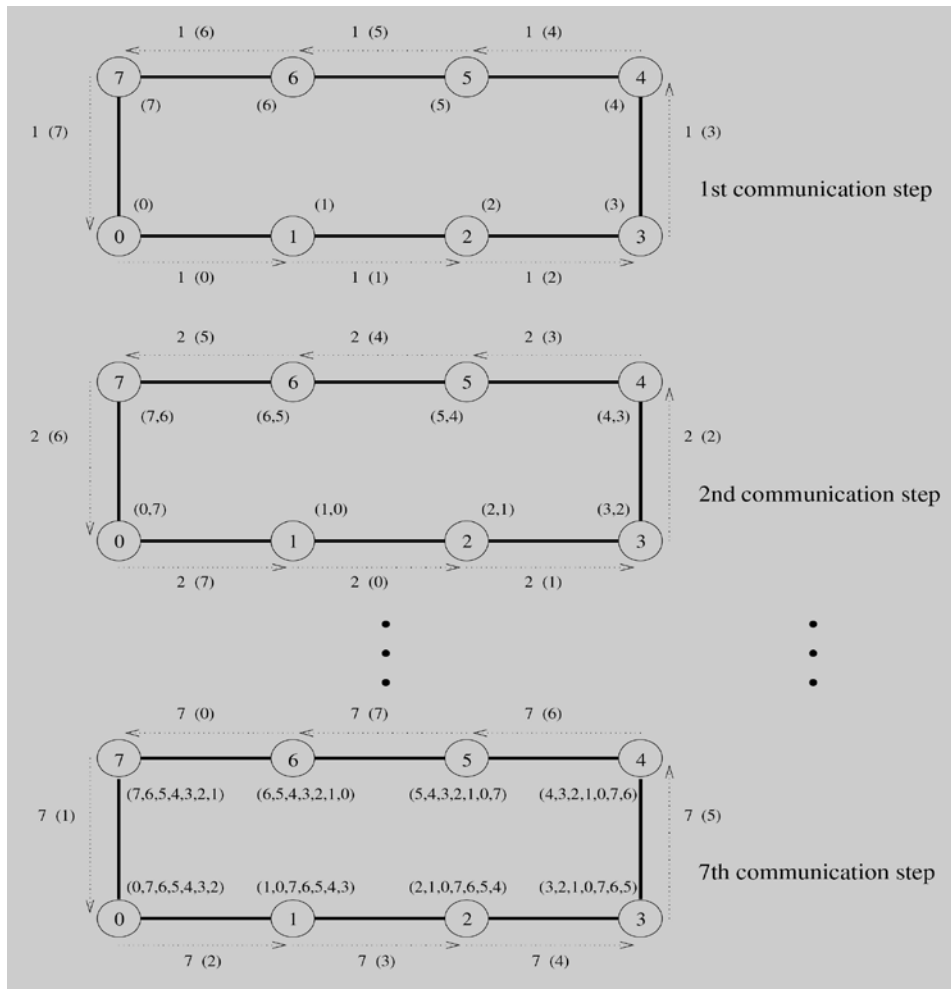
## Linear Array and Ring

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.

The following 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 the Figure, 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 9 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 . 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.**



Algorithm 4 gives a procedure for all-to-all broadcast on a  $p$ -node ring. The initial message to be broadcast is known locally as `my_msg` at each node. At the end of the procedure, each node stores the collection of all  $p$  messages in result. As the program shows, all-to-all broadcast on a mesh applies the linear array procedure twice, once along the rows and once along the columns.

**Algorithm 4 All-to-all broadcast on a  $p$ -node ring.**

```

1. procedure ALL_TO_ALL_BC_RING(my_id, my_msg, p, result)
2. begin
3.   left := (my_id - 1) mod p;
4.   right := (my_id + 1) mod p;

```

```

5.     result := my_msg;
6.     msg := result;
7.     for i := 1 to p - 1 do
8.         send msg to right;
9.         receive msg from left;
10.        result := result ∪ msg;
11.    endfor;
12. end ALL_TO_ALL_BC_RING

```

In all-to-all reduction, the dual of all-to-all broadcast, each node starts with  $p$  messages, each one destined to be accumulated at a distinct node. All-to-all reduction can be performed by reversing the direction and sequence of the messages. For example, the first communication step for all-to-all reduction on an 8-node ring would correspond to the last step of Figure with node 0 sending  $\text{msg}[1]$  to 7 instead of receiving it. The only additional step required is that upon receiving a message, a node must combine it with the local copy of the message that has the same destination as the received message before forwarding the combined message to the next neighbor. Algorithm 5 gives a procedure for all-to-all reduction on a  $p$ -node ring.

**Algorithm 5 All-to-all reduction on a  $p$ -node ring.**

```

1.  procedure ALL_TO_ALL_RED_RING(my_id, my_msg, p, result)
2.  begin
3.      left := (my_id - 1) mod p;
4.      right := (my_id + 1) mod p;
5.      recv := 0;
6.      for i := 1 to p - 1 do
7.          j := (my_id + i) mod p;
8.          temp := msg[j] + recv;
9.          send temp to left;
10.         receive recv from right;
11.     endfor;
12.     result := msg[my_id] + recv;
13. end ALL_TO_ALL_RED_RING

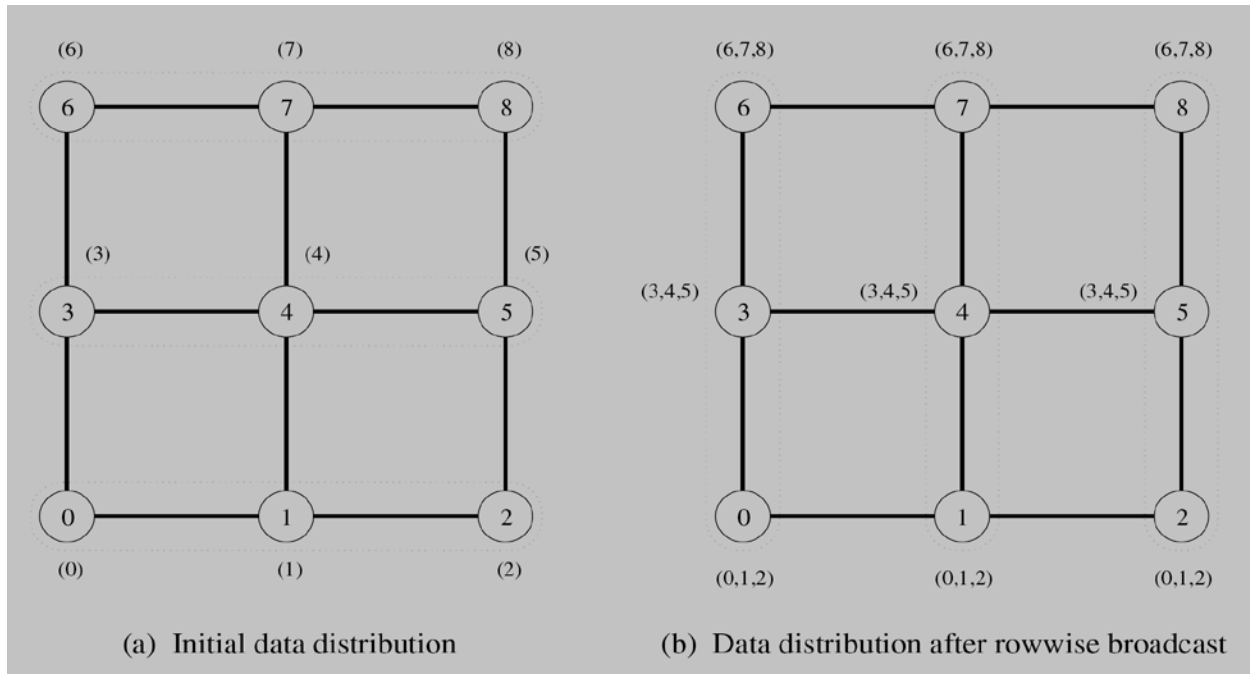
```

**Mesh**

Just like one-to-all broadcast, the all-to-all broadcast algorithm for the 2-D mesh is based on the linear array algorithm, treating rows and columns of the mesh as linear arrays. Once again, communication takes place in two phases. In the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the linear array. In this phase, all nodes collect  $\sqrt{p}$  messages corresponding to the  $\sqrt{p}$  nodes of their respective rows. Each node consolidates this information into a single message of size  $m\sqrt{p}$ , and proceeds to the second communication phase of the algorithm. The second communication phase is a columnwise all-to-all broadcast of the consolidated messages. By the end of this phase, each node obtains all  $p$  pieces of  $m$ -word data that originally resided on different nodes. The distribution of data among the nodes of a 3 x 3 mesh at the beginning of the first and the second phases of the algorithm is shown in Figure .

**Figure :** All-to-all broadcast on a 3 x 3 mesh. The groups of nodes communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all nodes get (0,1,2,3,4,5,6,7) (that is, a message from each node).





Algorithm 6 gives a procedure for all-to-all broadcast on a  $\sqrt{p} \times \sqrt{p}$  mesh.

**Algorithm 6 All-to-all broadcast on a square mesh of p nodes.**

```

1.  procedure ALL_TO_ALL_BC_MESH(my_id, my_msg, p, result)
2.  begin

/* Communication along rows */
3.    left := my_id - (my_id mod  $\sqrt{p}$ ) + (my_id - 1) mod  $\sqrt{p}$  ;
4.    right := my_id - (my_id mod  $\sqrt{p}$ ) + (my_id + 1) mod  $\sqrt{p}$  ;
5.    result := my_msg;
6.    msg := result;
7.    for i := 1 to  $\sqrt{p}$  - 1 do
8.      send msg to right;
9.      receive msg from left;
10.     result := result  $\cup$  msg;
11.   endfor;

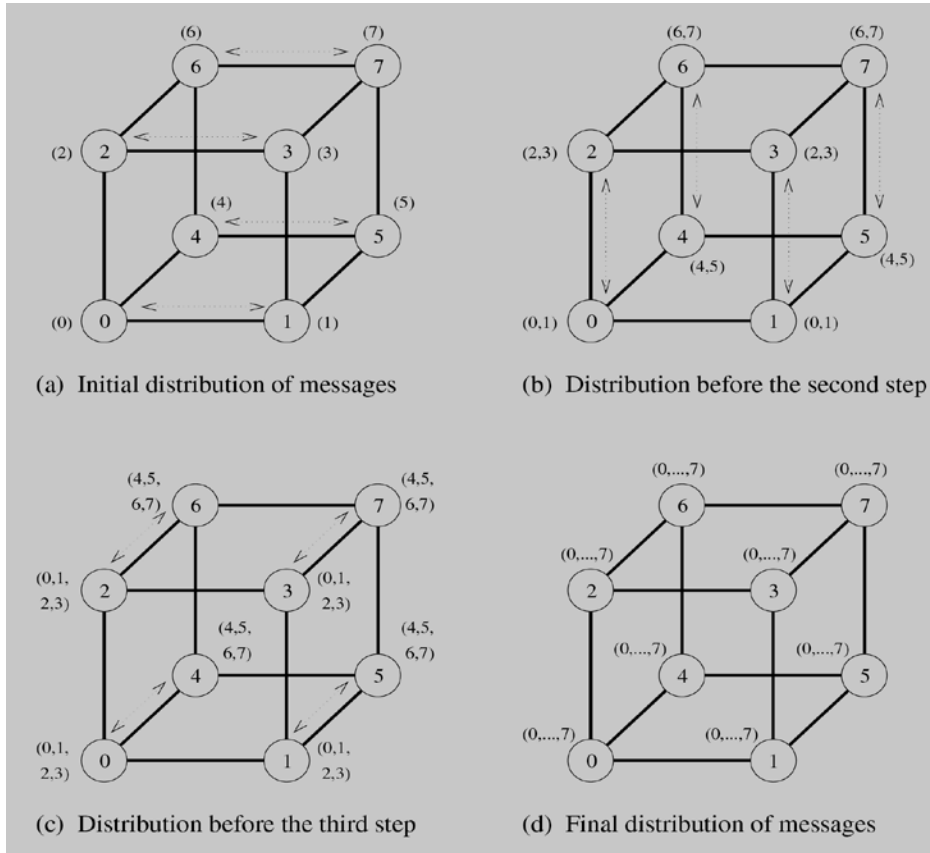
/* Communication along columns */
12.   up := (my_id -  $\sqrt{p}$ ) mod p;
13.   down := (my_id +  $\sqrt{p}$ ) mod p;
14.   msg := result;
15.   for i := 1 to  $\sqrt{p}$  - 1 do
16.     send msg to down;
17.     receive msg from up;
18.     result := result  $\cup$  msg;
19.   endfor;
20. end ALL_TO_ALL_BC_MESH

```

## Hypercube

The hypercube algorithm for all-to-all broadcast is an extension of the mesh algorithm to  $\log p$  dimensions. The procedure requires  $\log p$  steps. Communication takes place along a different dimension of the  $p$ -node hypercube in each step. In every step, pairs of nodes exchange their data and double the size of the message to be transmitted in the next step by concatenating the received message with their current data. The following figure shows these steps for an eight-node hypercube with bidirectional communication channels.

Figure . All-to-all broadcast on an eight-node hypercube.



Algorithm 7 gives a procedure for implementing all-to-all broadcast on a  $d$ -dimensional hypercube. Communication starts from the lowest dimension of the hypercube and then proceeds along successively higher dimensions (Line 4). In each iteration, nodes communicate in pairs so that the labels of the nodes communicating with each other in the  $i$ th iteration differ in the  $i$ th least significant bit of their binary representations (Line 5). After an iteration's communication steps, each node concatenates the data it receives during that iteration with its resident data (Line 8). This concatenated message is transmitted in the following iteration.

Algorithm 7 All-to-all broadcast on a  $d$ -dimensional hypercube.

```

1.  procedure ALL_TO_ALL_BC_HCUBE(my_id, my_msg, d, result)
2.  begin
3.      result := my_msg;
4.      for i := 0 to d - 1 do

```

```

5.         partner := my_id XOR 2i;
6.         send result to partner;
7.         receive msg from partner;
8.         result := result ∪ msg;
9.     endfor;
10. end ALL_TO_ALL_BC_HCUBE

```

As usual, the algorithm for all-to-all reduction can be derived by reversing the order and direction of messages in all-to-all broadcast. Furthermore, instead of concatenating the messages, the reduction operation needs to select the appropriate subsets of the buffer to send out and accumulate received messages in each iteration. Algorithm 8 gives a procedure for all-to-all reduction on a d-dimensional hypercube. It uses `senloc` to index into the starting location of the outgoing message and `recloc` to index into the location where the incoming message is added in each iteration.

**Algorithm 8 All-to-all broadcast on a d-dimensional hypercube. AND and XOR are bitwise logical-and and exclusive-or operations, respectively.**

```

1.  procedure ALL_TO_ALL_RED_HCUBE(my_id, msg, d, result)
2.  begin
3.      recloc := 0;
4.      for i := d - 1 to 0 do
5.          partner := my_id XOR 2i;
6.          j := my_id AND 2i;
7.          k := (my_id XOR 2i) AND 2i;
8.          senloc := recloc + k;
9.          recloc := recloc + j;
10.         send msg[senloc .. senloc + 2i - 1] to partner;
11.         receive temp[0 .. 2i - 1] from partner;
12.         for j := 0 to 2i - 1 do
13.             msg[recloc + j] := msg[recloc + j] + temp[j];
14.         endfor;
15.     endfor;
16.     result := msg[my_id];
17. end ALL_TO_ALL_RED_HCUBE

```

## 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.

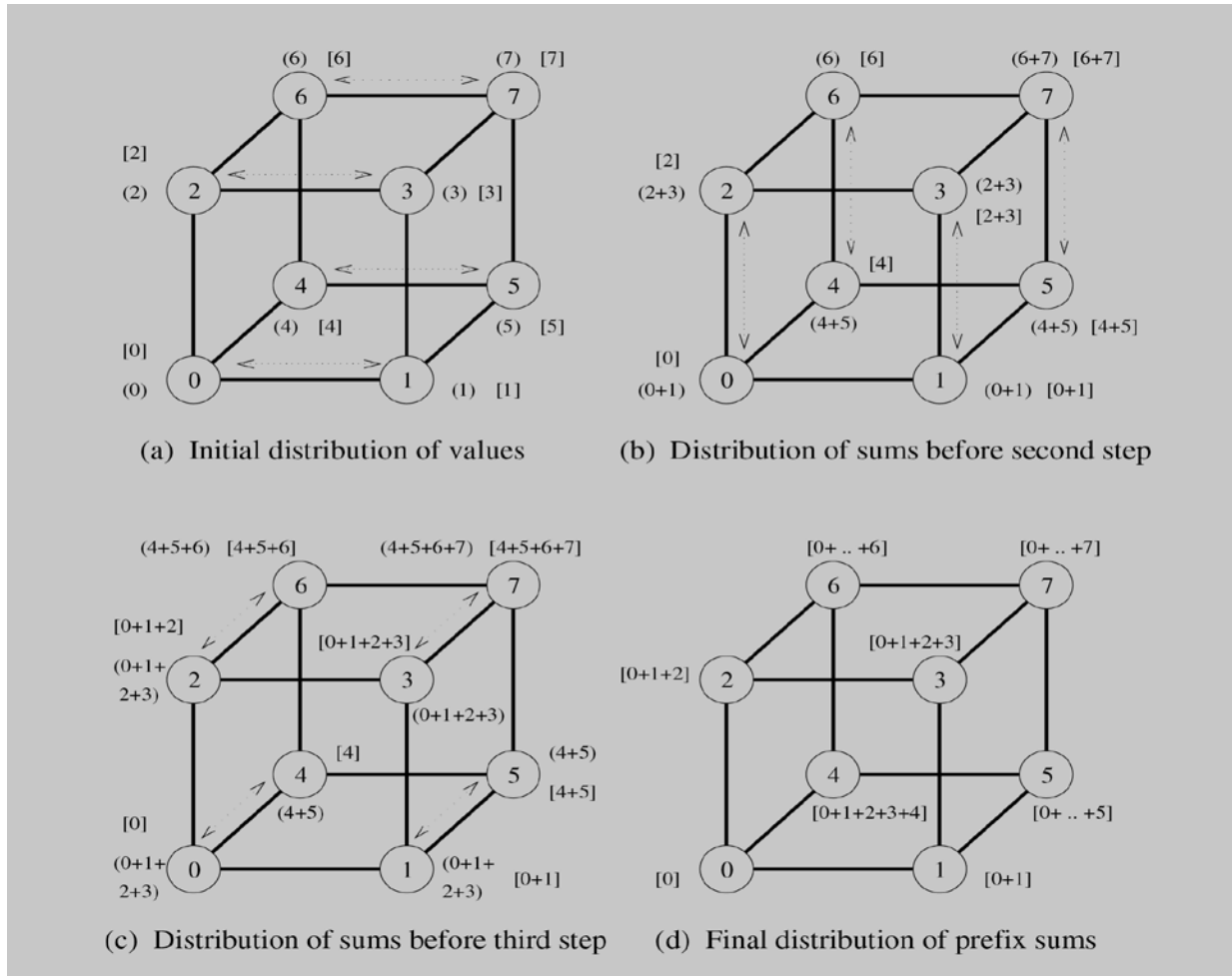
A simple method to perform all-reduce is to perform an all-to-one reduction followed by a one-to-all broadcast. However, there is a faster way to perform all-reduce by using the communication pattern of all-to-all broadcast. Assume that each integer in parentheses in the figure, instead of denoting a 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  $(0 + 1 + 2 + \dots + 7)$  (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  $T = (t_s + t_w m) \log p$ .

Algorithm 7 can be used to perform a sum of  $p$  numbers if `my_msg`, `msg`, and `result` are numbers (rather than messages), and the union operation (`'∪'`) on Line 8 is replaced by addition.

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  $n_0, n_1, \dots, n_{p-1}$  (one on each node), the problem is to compute the sums  $s_k = \sum_{i=0}^k n_i$  for all  $k$  between 0 and  $p - 1$ . For example, if the original sequence of numbers is  $\langle 3, 1, 4, 0, 2 \rangle$ , then the sequence of prefix sums is  $\langle 3, 4, 8, 8, 10 \rangle$ . 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.

The following figure illustrates the prefix sums procedure for an eight-node hypercube. This figure is a modification of the above figure. 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 the Figure. 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 :** 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.



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, although the presence or absence of these messages does not affect the results of the algorithm. Algorithm 9 gives a procedure to solve the prefix sums problem on a  $d$ -dimensional hypercube.

**Algorithm 9 Prefix sums on a  $d$ -dimensional hypercube.**

```

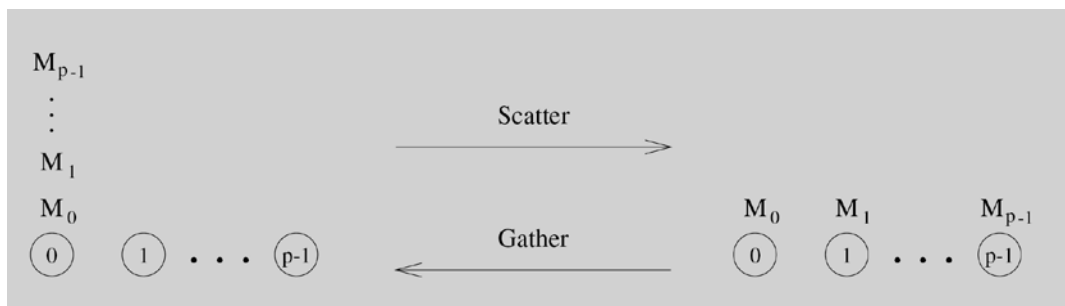
1.  procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
2.  begin
3.      result := my_number;
4.      msg := result;
5.      for i := 0 to d - 1 do
6.          partner := my_id XOR 2i;
7.          send msg to partner;
8.          receive number from partner;
9.          msg := msg + number;
10.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

```

**Scatter and Gather**

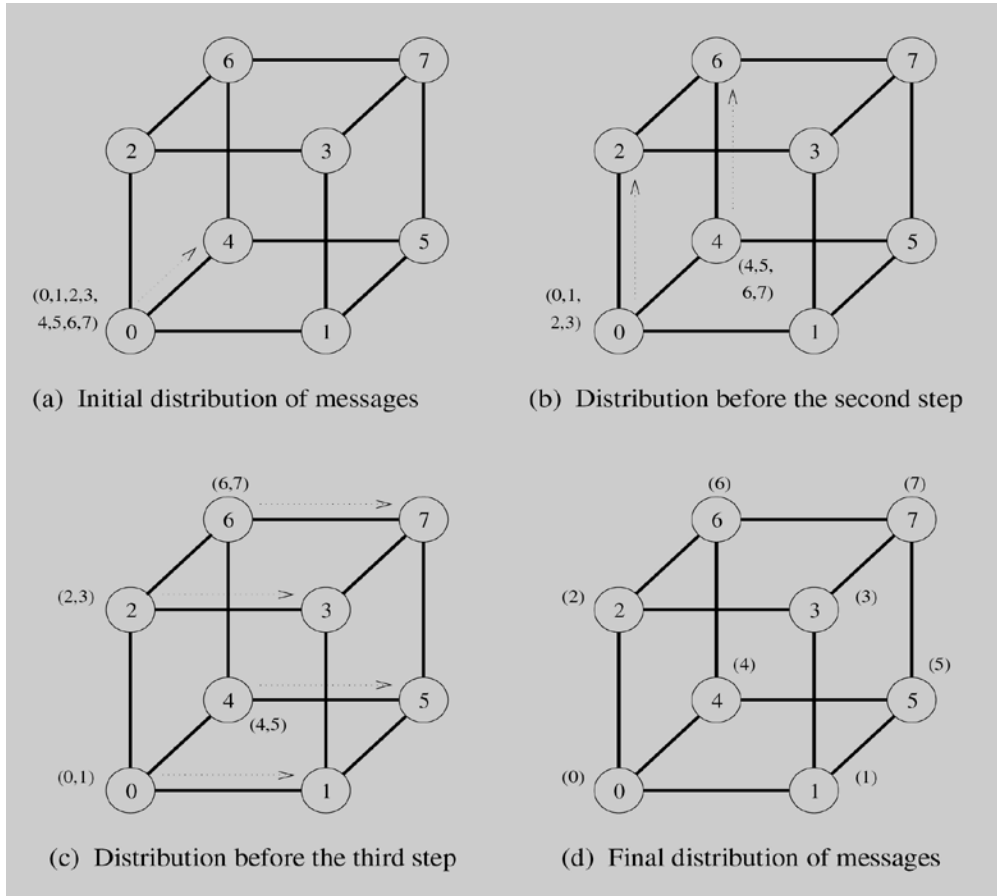
In the scatter operation, a single node sends a unique message of size  $m$  to every other node. This operation is also known as one-to-all personalized communication. One-to-all personalized communication is different from one-to-all broadcast in that the source node starts with  $p$  unique messages, 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 : 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 shows the communication steps for the scatter operation on an eight-node hypercube. The communication patterns of one-to-all broadcast and scatter are identical. Only the size and the contents of messages are different. In the figure 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 : The scatter operation on an eight-node hypercube.**



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

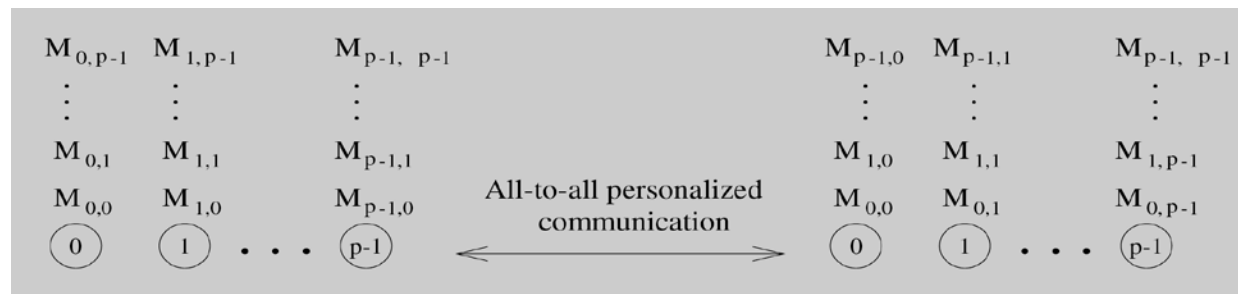
The scatter and gather operations can also be performed on a linear array and on a 2-D square mesh in time  $t_s \log p + t_w m(p - 1)$ . Note that disregarding the term due to message-startup time, the cost of scatter and gather operations for large messages on any  $k$ -d mesh interconnection network 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.

### 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 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 : All-to-all personalized communication.

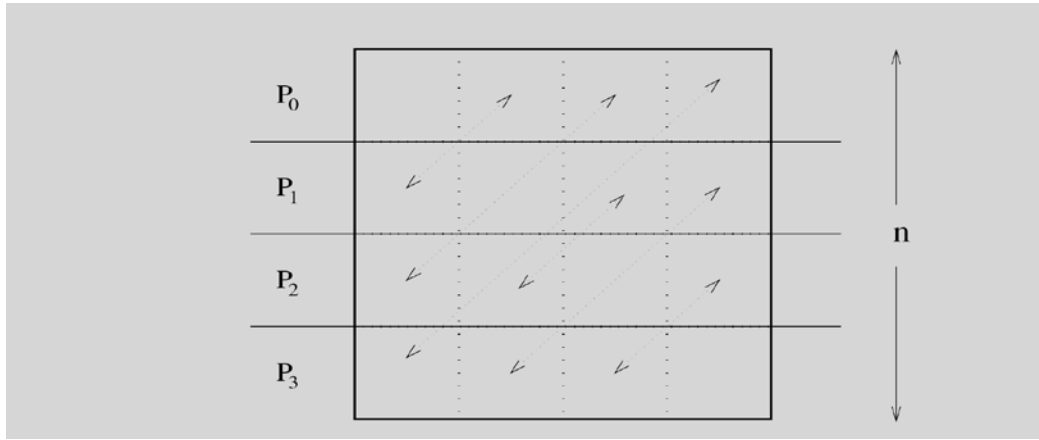


Example : Matrix transposition

The transpose of an  $n \times n$  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  $n \times n$  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  $[i, 0], [i, 1], \dots, [i, n - 1]$ . 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 for a  $4 \times 4$  matrix mapped onto four processes using one-dimensional rowwise 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 : All-to-all personalized communication in transposing a  $4 \times 4$  matrix using four processes.





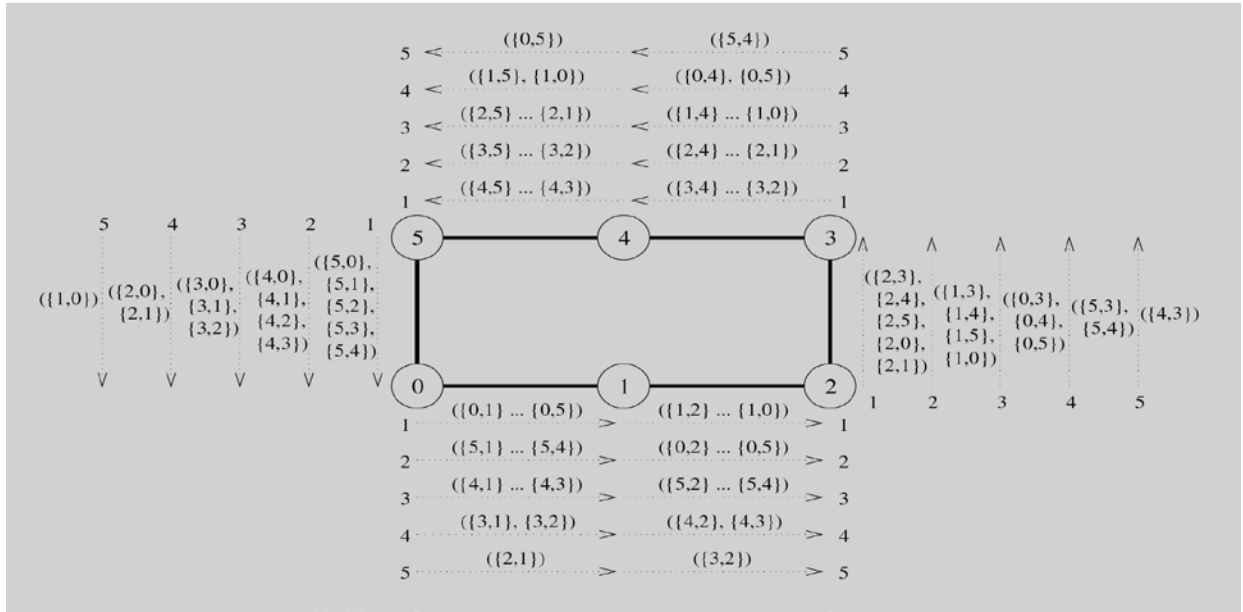
In general, if we use  $p$  processes such that  $p \leq n$ , then each process initially holds  $n/p$  rows (that is,  $n^2/p$  elements) of the matrix. Performing the transposition now involves an all-to-all personalized communication of matrix blocks of size  $n/p \times n/p$ , instead of individual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

We now discuss the implementation of all-to-all personalized communication on parallel computers with linear array, mesh, and hypercube interconnection networks. The communication patterns of all-to-all personalized communication are identical to those of all-to-all broadcast on all three architectures. Only the size and the contents of messages are different.

### Ring

Figure 4.18 shows the steps in an all-to-all personalized communication on a six-node linear array. To perform this operation, every node sends  $p - 1$  pieces of data, each of size  $m$ . In the figure, these pieces of data are identified by pairs of integers of the form  $\{i, j\}$ , where  $i$  is the source of the message and  $j$  is its final destination. First, each node sends all pieces of data as one consolidated message of size  $m(p - 1)$  to one of its neighbors (all nodes communicate in the same direction). Of the  $m(p - 1)$  words of data received by a node in this step, one  $m$ -word packet belongs to it. Therefore, each node extracts the information meant for it from the data received, and forwards the remaining  $(p - 2)$  pieces of size  $m$  each to the next node. This process continues for  $p - 1$  steps. The total size of data being transferred between nodes decreases by  $m$  words in each successive step. In every step, each node adds to its collection one  $m$ -word packet originating from a different node. Hence, in  $p - 1$  steps, every node receives the information from all other nodes in the ensemble.

**Figure :** All-to-all personalized communication on a six-node ring. The label of each message is of the form  $\{x, y\}$ , where  $x$  is the label of the node that originally owned the message, and  $y$  is the label of the node that is the final destination of the message. The label  $(\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\})$  indicates a message that is formed by concatenating  $n$  individual messages.



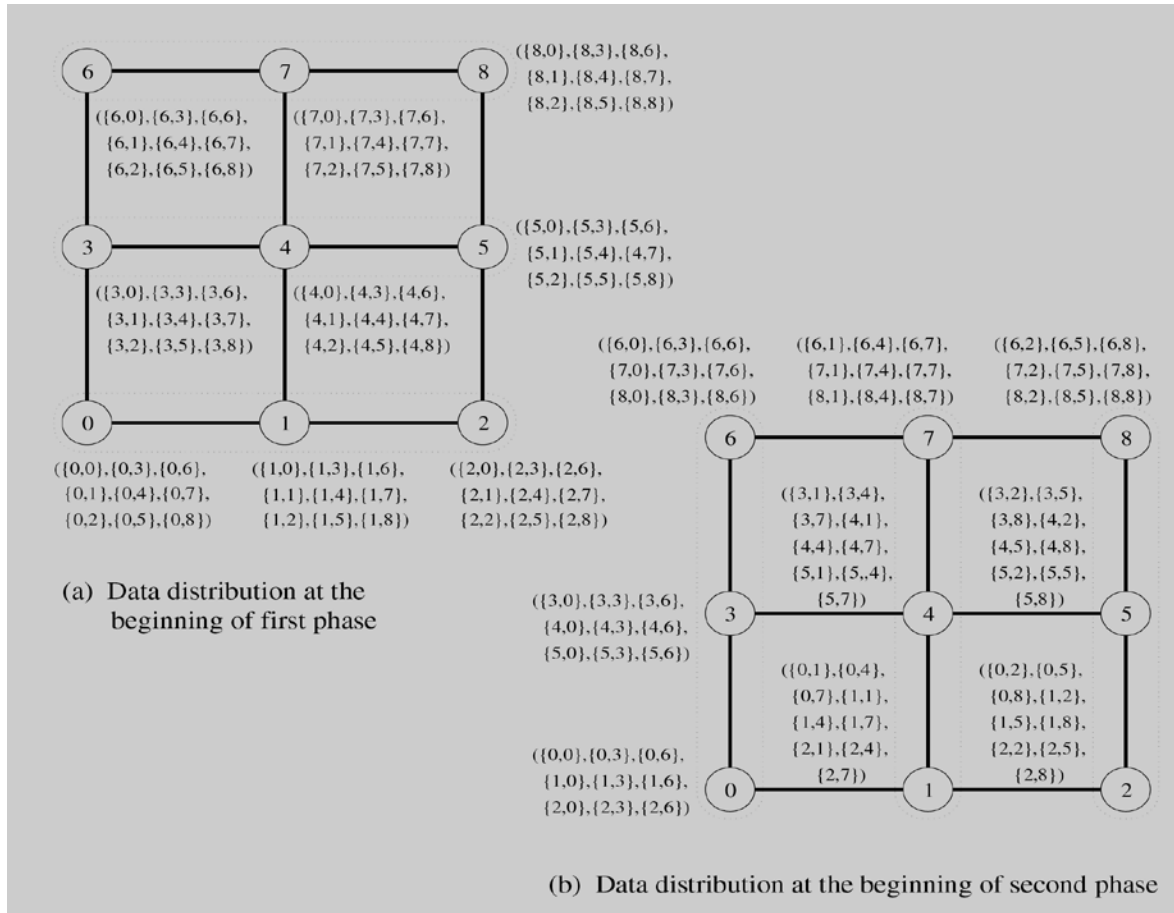
In the above procedure, all messages are sent in the same direction. If half of the messages are sent in one direction and the remaining half are sent in the other direction, then the communication cost due to the  $t_w$  can be reduced by a factor of two. For the sake of simplicity, we ignore this constant-factor improvement.

**Cost Analysis** On a ring or a bidirectional linear array, all-to-all personalized communication involves  $p - 1$  communication steps. Since the size of the messages transferred in the  $i$ th step is  $m(p - i)$ , the total time taken by this operation is  $T = \sum_{i=1}^{p-1} (t_s + t_w m(p - i))$

### Mesh

In all-to-all personalized communication on a  $\sqrt{p} \times \sqrt{p}$  mesh, each node first groups its  $p$  messages according to the columns of their destination nodes. Figure 4.19 shows a  $3 \times 3$  mesh, in which every node initially has nine  $m$ -word messages, one meant for each node. Each node assembles its data into three groups of three messages each (in general,  $\sqrt{p}$  groups of  $\sqrt{p}$  messages each). The first group contains the messages destined for nodes labeled 0, 3, and 6; the second group contains the messages for nodes labeled 1, 4, and 7; and the last group has messages for nodes labeled 2, 5, and 8.

**Figure :** The distribution of messages at the beginning of each phase of all-to-all personalized communication on a  $3 \times 3$  mesh. At the end of the second phase, node  $i$  has messages  $\{(0, i), \dots, \{8, i)\}$ , where  $0 \leq i \leq 8$ . The groups of nodes communicating together in each phase are enclosed in dotted boundaries.



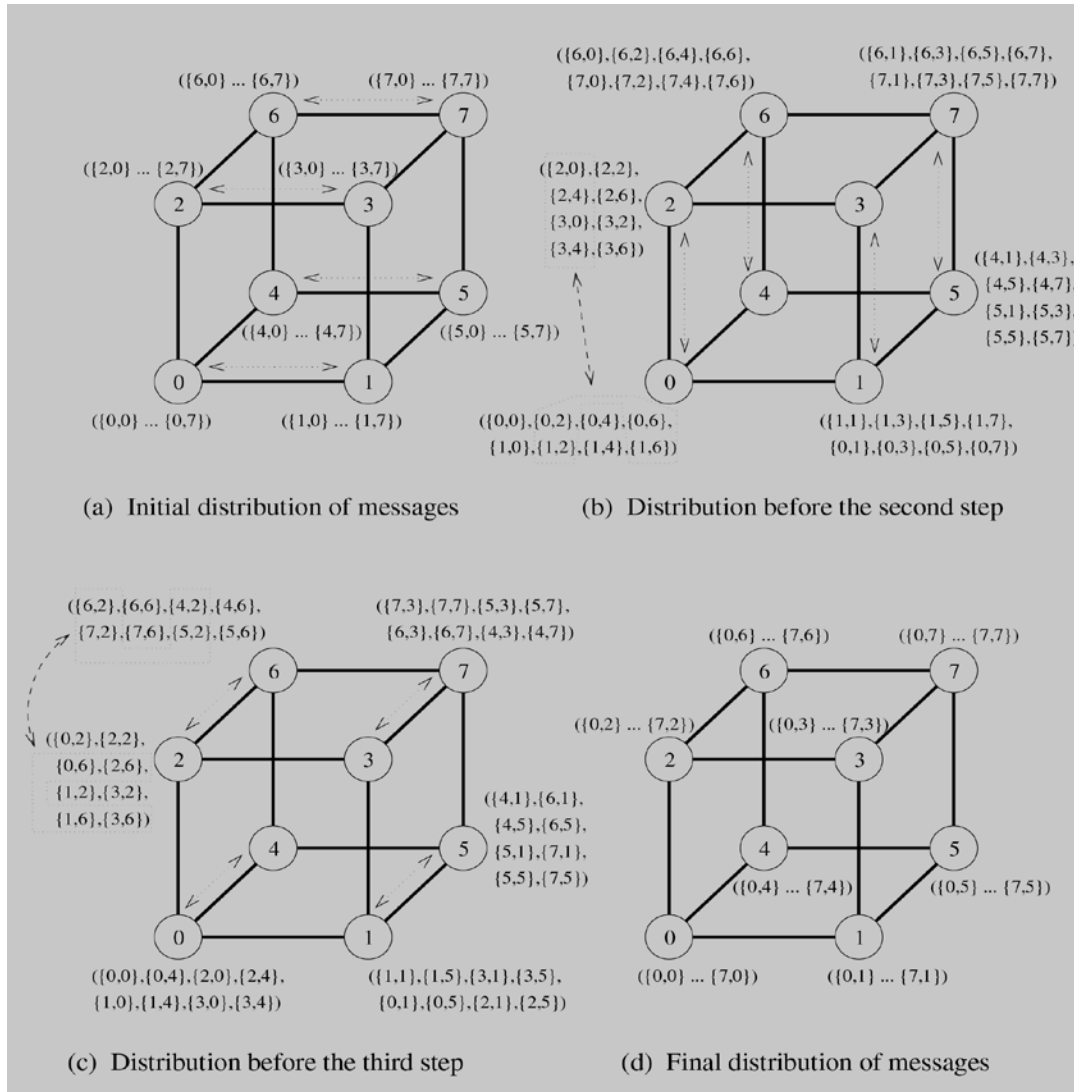
After the messages are grouped, all-to-all personalized communication is performed independently in each row with clustered messages of size  $m\sqrt{p}$ . One cluster contains the information for all  $\sqrt{p}$  nodes of a particular column.

Before the second communication phase, the messages in each node are sorted again, this time according to the rows of their destination nodes; then communication similar to the first phase takes place in all the columns of the mesh. By the end of this phase, each node receives a message from every other node.

### Hypercube

One way of performing all-to-all personalized communication on a  $p$ -node hypercube is to simply extend the two-dimensional mesh algorithm to  $\log p$  dimensions. Figure shows the communication steps required to perform this operation on a three-dimensional hypercube. As shown in the figure, communication takes place in  $\log p$  steps. Pairs of nodes exchange data in a different dimension in each step. Recall that in a  $p$ -node hypercube, a set of  $p/2$  links in the same dimension connects two subcubes of  $p/2$  nodes each. At any stage in all-to-all personalized communication, every node holds  $p$  packets of size  $m$  each. While communicating in a particular dimension, every node sends  $p/2$  of these packets (consolidated as one message). The destinations of these packets are the nodes of the other subcube connected by the links in current dimension.

Figure . An all-to-all personalized communication algorithm on a three-dimensional hypercube.



In the preceding procedure, a node must rearrange its messages locally before each of the  $\log p$  communication steps. This is necessary to make sure that all  $p/2$  messages destined for the same node in a communication step occupy contiguous memory locations so that they can be transmitted as a single consolidated message.

**Cost Analysis** In the above hypercube algorithm for all-to-all personalized communication,  $mp/2$  words of data are exchanged along the bidirectional channels in each of the  $\log p$  iterations. The resulting total communication time is  $T = \sum_{i=1}^{p-1} (t_s + t_w mp / 2) \log p$ .

Before each of the  $\log p$  communication steps, a node rearranges  $mp$  words of data. Hence, a total time of  $t_r mp \log p$  is spent by each node in local rearrangement of data during the entire procedure. Here  $t_r$  is the time needed to perform a read and a write operation on a single word of data in a node's local memory. For most practical computers,  $t_r$  is much smaller than  $t_w$ ; hence, the time to perform an all-to-all personalized communication is dominated by the communication time.

### Analytical Modeling of Parallel Programs



A **parallel system** is the combination of an algorithm and the parallel architecture on which it is implemented.

## Sources of Overhead in Parallel Programs

Using twice as many hardware resources, one can reasonably expect a program to run twice as fast. However, in typical parallel programs, this is rarely the case, due to a variety of overheads associated with parallelism. An accurate quantification of these overheads is critical to the understanding of parallel program performance.

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

**Idling:** Processing elements in a parallel system may become idle 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 predict the size of the subtasks assigned to various processing elements. Hence, the problem cannot be subdivided statically among 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 certain points during parallel program execution. If all processing elements are not ready for synchronization at the same time, then the ones that are ready sooner 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 fastest 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.

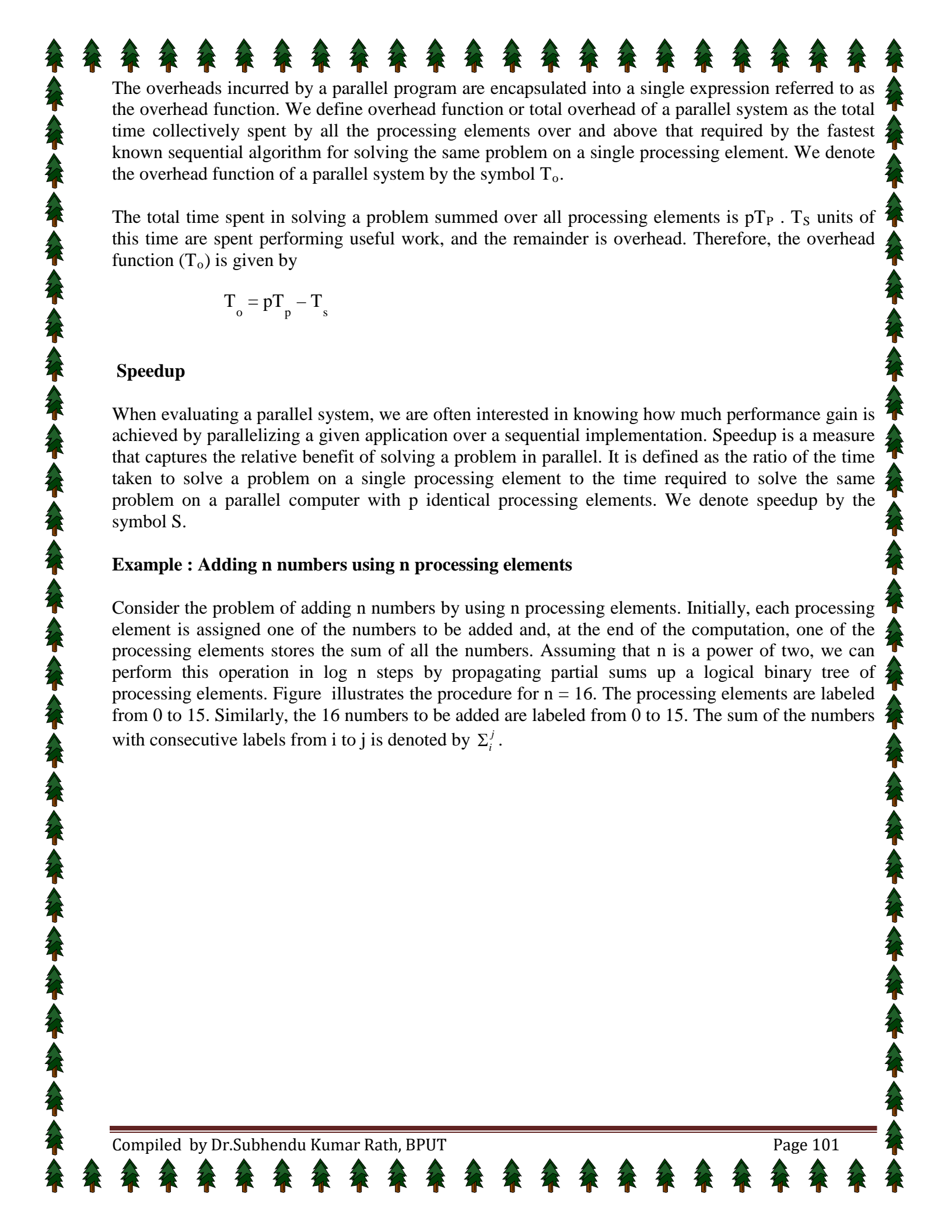
## Performance Metrics for Parallel Systems

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

### Execution Time

The serial runtime of a program is the time elapsed 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  $T_P$ .

### Total Parallel Overhead



The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function. We define overhead function or total overhead of a parallel system as the total time collectively spent 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 over 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

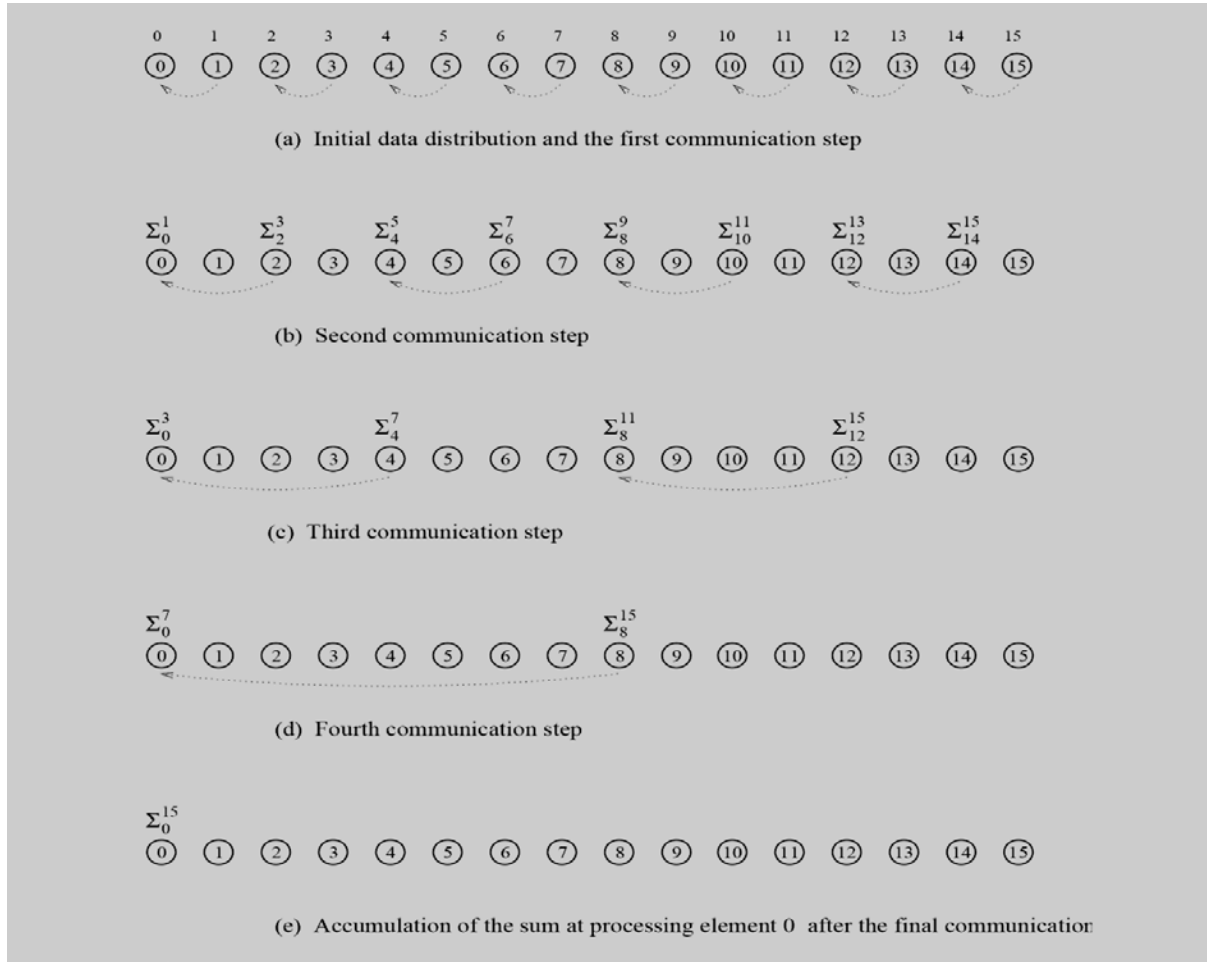
$$T_o = pT_p - T_s$$

### Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined 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 denote speedup by the symbol  $S$ .

### Example : 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 partial sums up a logical binary tree of processing elements. Figure 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  $\Sigma_i^j$ .



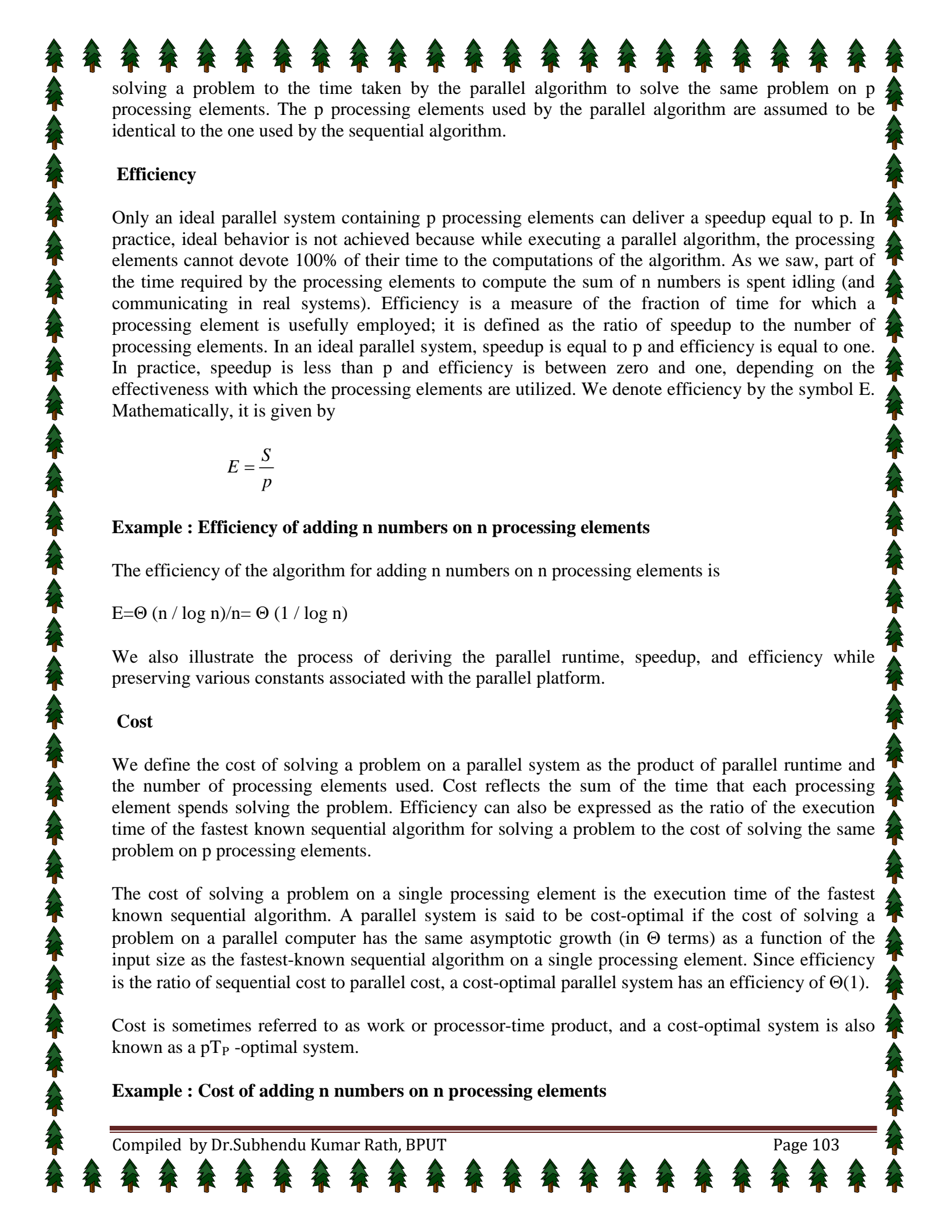
Each step consists of one addition and the communication of a single word. The addition can be performed in some constant time, say  $t_c$ , and the communication of a single word can be performed in time  $t_s + t_w$ . Therefore, the addition and communication operations take a constant amount of time.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

$$T_p = \Theta(\log n)$$

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

$$S = \Theta(n / \log n)$$

For a given 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 least amount of time. Given a parallel algorithm, it is fair 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 solve a problem is not known, or its runtime has a large 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 best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same 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 same problem on  $p$  processing elements. The  $p$  processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

### 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, 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

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

### Example : Efficiency of adding $n$ numbers on $n$ processing elements

The efficiency of the algorithm for adding  $n$  numbers on  $n$  processing elements is

$$E = \Theta(n / \log n) / n = \Theta(1 / \log n)$$

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

### Cost

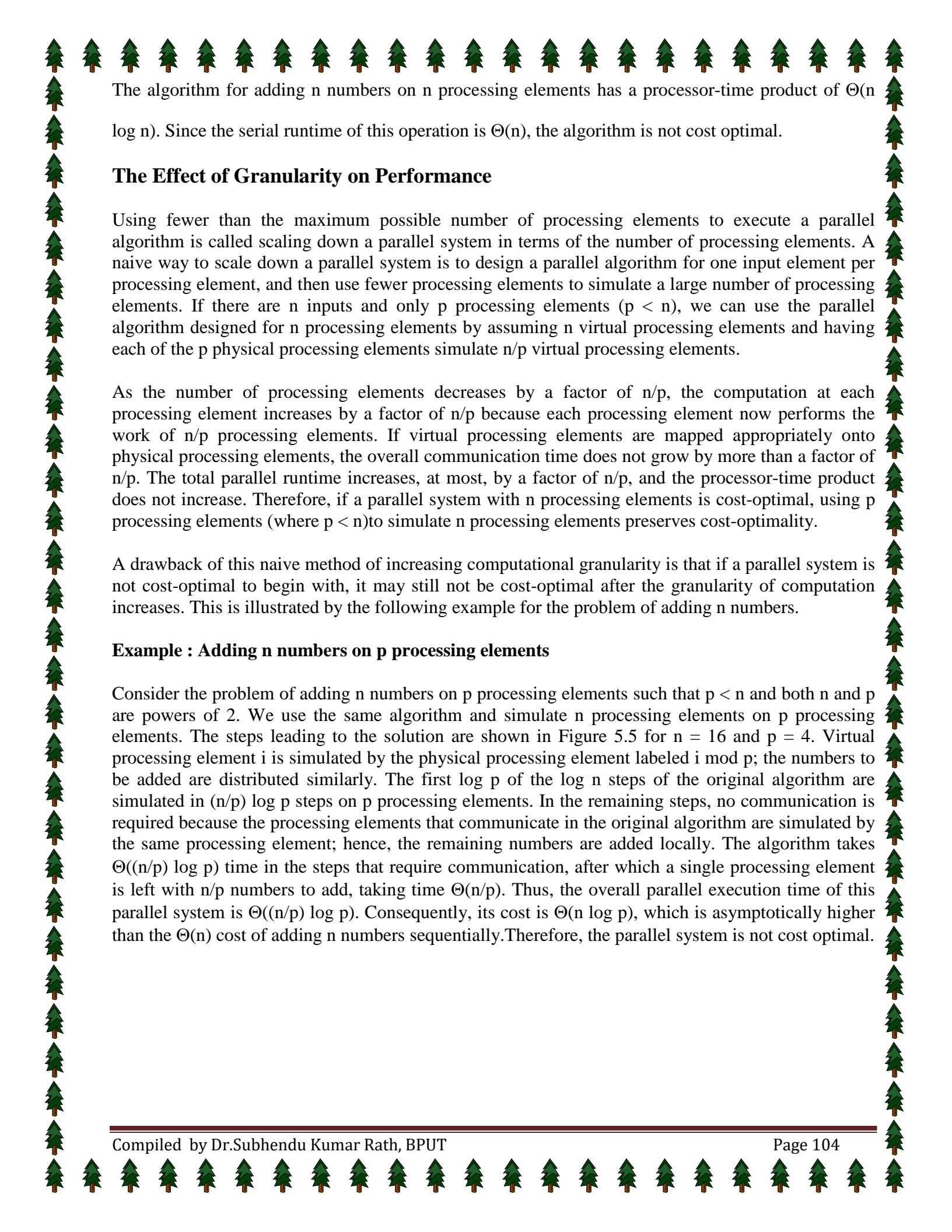
We define the cost of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of 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  $\Theta$  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  $\Theta(1)$ .

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.

### Example : Cost of adding $n$ numbers on $n$ processing elements





The algorithm for adding  $n$  numbers on  $n$  processing elements has a processor-time product of  $\Theta(n \log n)$ . Since the serial runtime of this operation is  $\Theta(n)$ , the algorithm is not cost optimal.

### The Effect of Granularity on Performance

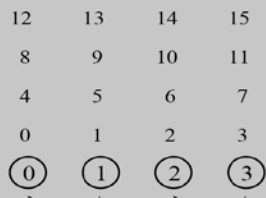
Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called scaling down a parallel system in terms of the number of processing elements. A naive way to scale down a parallel system is to design a parallel algorithm for one input element per processing 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.

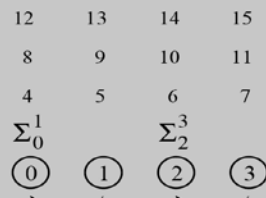
A drawback of this naive method of increasing computational granularity is that if a parallel system is not cost-optimal to begin with, it may still not be cost-optimal after the granularity of computation increases. This is illustrated by the following example for the problem of adding  $n$  numbers.

#### Example : Adding $n$ numbers on $p$ processing elements

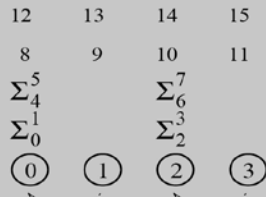
Consider the problem of adding  $n$  numbers on  $p$  processing elements such that  $p < n$  and both  $n$  and  $p$  are powers of 2. We use the same algorithm and simulate  $n$  processing elements on  $p$  processing elements. The steps leading to the solution are shown in Figure 5.5 for  $n = 16$  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.



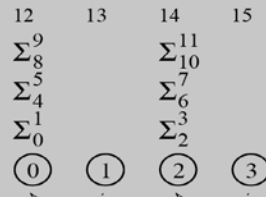
Substep 1



Substep 2

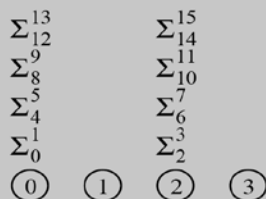


Substep 3

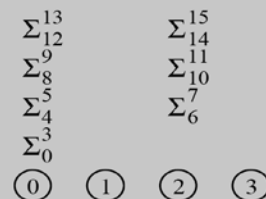


Substep 4

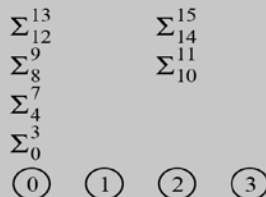
(a) Four processors simulating the first communication step of 16 processors



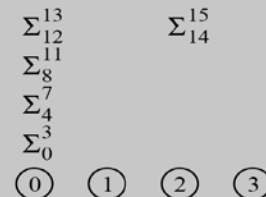
Substep 1



Substep 2

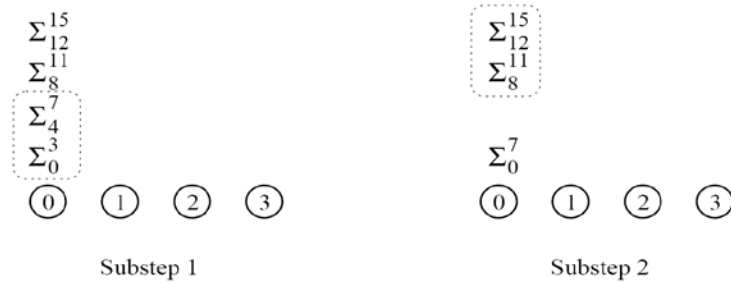


Substep 3



Substep 4

(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

It has been 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, 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 (Figure (c) and (d)) did not require any communication. However, this reduction in communication was not enough to make the algorithm cost-optimal. The following Example 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.

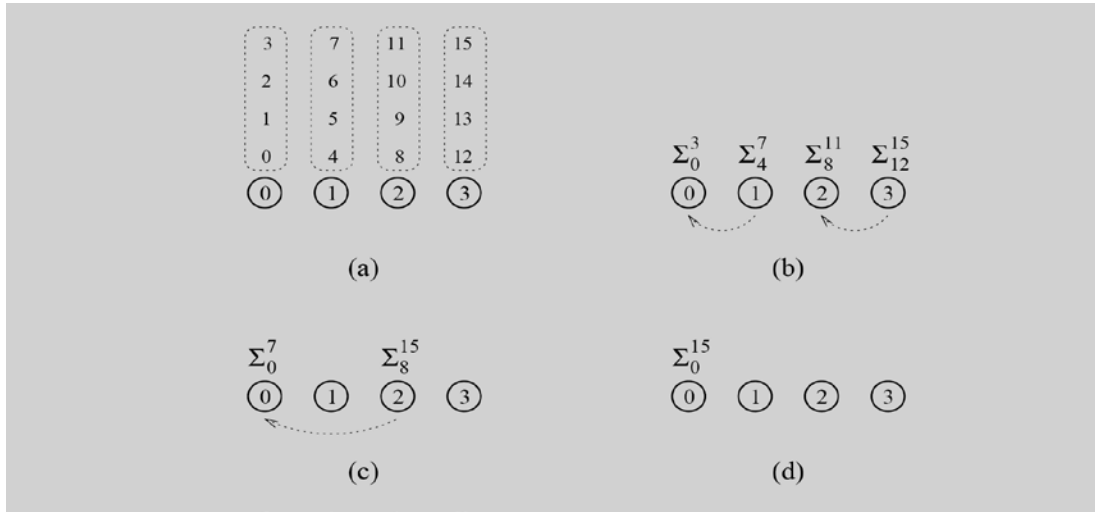
### Example : Adding $n$ numbers cost-optimally

An alternate method for adding  $n$  numbers using  $p$  processing elements is illustrated 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  $\Theta(\log p)$ . The parallel runtime of this algorithm is

$$T_p = \Theta(n/p + \log p)$$

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: 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.

### 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.

### 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  $T_o = pT_p - T_s$ , we can rewrite this expression as

$$E = \frac{T_s}{T_o + T_s} = \frac{1}{1 + \frac{T_o}{T_s}}$$

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  $t_{\text{serial}}$ , then during this time all the other processing elements must be idle. This corresponds to a total overhead function of  $(p - 1) \times t_{\text{serial}}$ . 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 superlinearly in the number of processing elements.

### Example : Speedup and efficiency as functions of the number of processing elements

Consider the problem of adding  $n$  numbers on  $p$  processing elements. However, to illustrate actual speedups, we work with constants here instead of asymptotic. 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:

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

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

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

These expressions can be used to calculate the speedup and efficiency for any pair of  $n$  and  $p$ .

### 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.

**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. 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  $W$ .

### 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:

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

The resulting expression for speedup is

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

Finally, we write the expression for efficiency as

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

In this equation, 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 the equation is maintained at a constant value. For a desired value  $E$  of efficiency,

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

$$\Rightarrow T_o(W, p) / W = \frac{1 - E}{E}$$

$$\Rightarrow W = \frac{E}{1 - E} T_o(W, p)$$

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

$$W = KT_o(W, p)$$

From the above equation, 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.

### Cost-Optimality and the Isoefficiency Function

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

$$pT_p = \Theta(W)$$

Substituting the expression for  $T_p$  from the right-hand side of  $T_p = \frac{W + T_o(W, p)}{p}$ , we get the following:

$$W + T_o(W, p) = \Theta(W)$$

$$\Rightarrow T_o(W, p) = O(W)$$

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

The above Equations suggest that a parallel system is cost-optimal if and only if its overhead function does not asymptotically exceed the problem size.

### 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. 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:

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

Let  $p_0$  be the value of the number of processing elements that satisfies the above equation. 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 : Minimum execution time for adding $n$ numbers

The parallel run time for the problem of adding  $n$  numbers on  $p$  processing elements can be approximated by

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

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

$$-\frac{n}{p^2} + \frac{2}{p} = 0$$

$$\Rightarrow -n + 2p = 0$$

$$\Rightarrow p = \frac{n}{2}$$

Substituting  $p = n/2$  in the equation, we get

$$T_p^{\min} = 2 \log n$$

In the above example, 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  $T_p^{\text{cost-opt}}$  be the minimum time in which a problem can be solved by a cost-optimal parallel system. 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  $W = \Omega(f(p))$ . 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, the lower bound on the parallel runtime for solving a problem of size  $W$  cost-optimally is

$$T_p^{\text{cost-opt}} = \Omega\left(\frac{W}{f^{-1}(W)}\right)$$

Example : Minimum cost-optimal execution time for adding  $n$  numbers

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,  $\log n \cong \log p$ . If  $n = f(p) = p \log p$ , then  $p = f^{-1}(n) = n/\log p \cong n/\log n$ . 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 the above equation, we get

$$T_p^{\text{cost-opt}} = \log n + \log\left(\frac{n}{\log n}\right) = 2 \log n - \log \log n$$

It is interesting to observe that both  $T_p^{\min}$  and  $T_p^{\text{cost-opt}}$  for adding  $n$  numbers are  $\Theta(\log n)$ .

## Matrix-Vector Multiplication

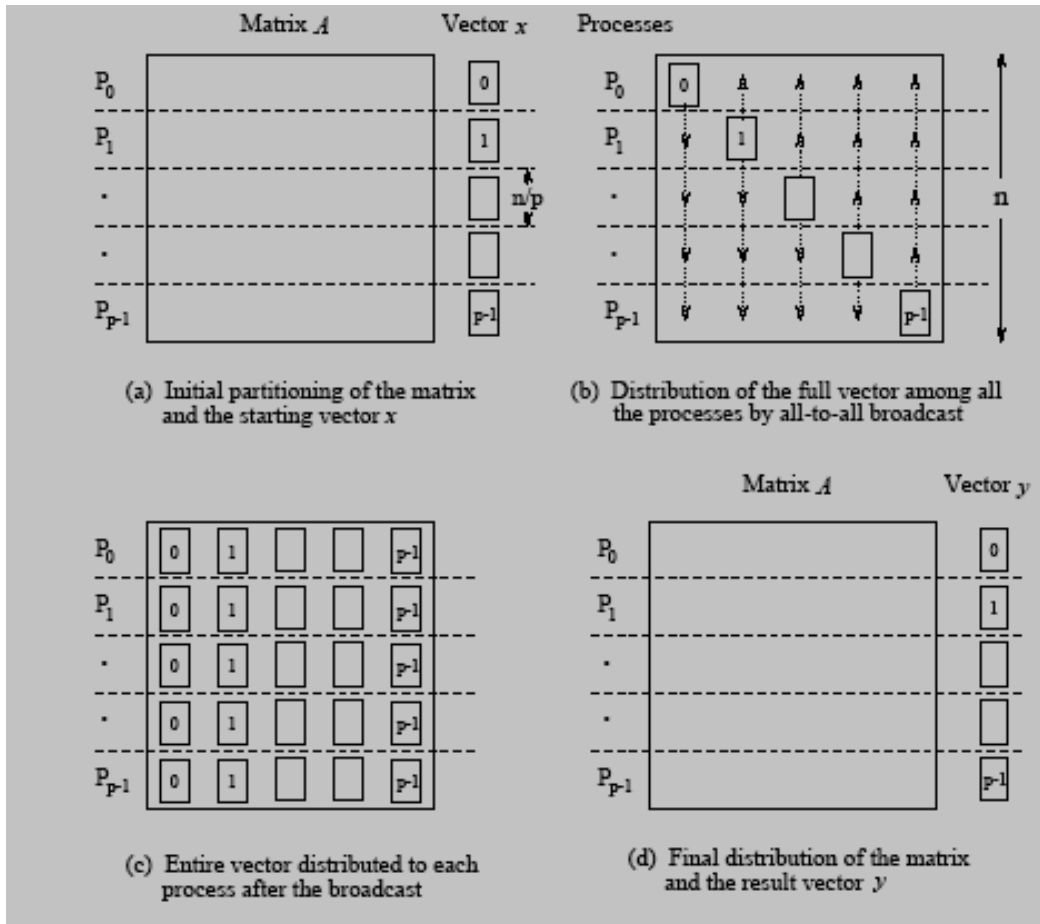
- We aim to multiply a dense  $n \times n$  matrix  $A$  with an  $n \times 1$  vector  $x$  to yield the  $n \times 1$  result vector  $y$ .
- The serial algorithm requires  $n^2$  multiplications and additions.  
 $W = n^2$ .
- The  $n \times n$  matrix is partitioned among  $n$  processors, with each processor storing complete row of the matrix.
- The  $n \times 1$  vector  $x$  is distributed such that each process owns one of its elements.

## Rowwise 1-D Partitioning

- Since each process starts with only one element of  $x$ , an all-to-all broadcast is required to distribute all the elements to all the processes.
- Process  $P_i$  now computes
 
$$y[i] = \sum_{j=0}^{n-1} (A[i, j] \times x[j])$$
- The all-to-all broadcast and the computation of  $y[i]$  both take time  $\Theta(n)$ . Therefore, the parallel time is  $\Theta(n)$ .
- Consider now the case when  $p < n$  and we use block 1D partitioning.
- Each process initially stores  $n/p$  complete rows of the matrix and a portion of the vector of size  $n/p$ .
- The all-to-all broadcast takes place among  $p$  processes and involves messages of size  $n/p$ .
- This is followed by  $n/p$  local dot products.
- Thus, the parallel run time of this procedure is

$$T_p = \frac{n^2}{p} + t_s \log p + t_w n$$

This is cost-optimal.



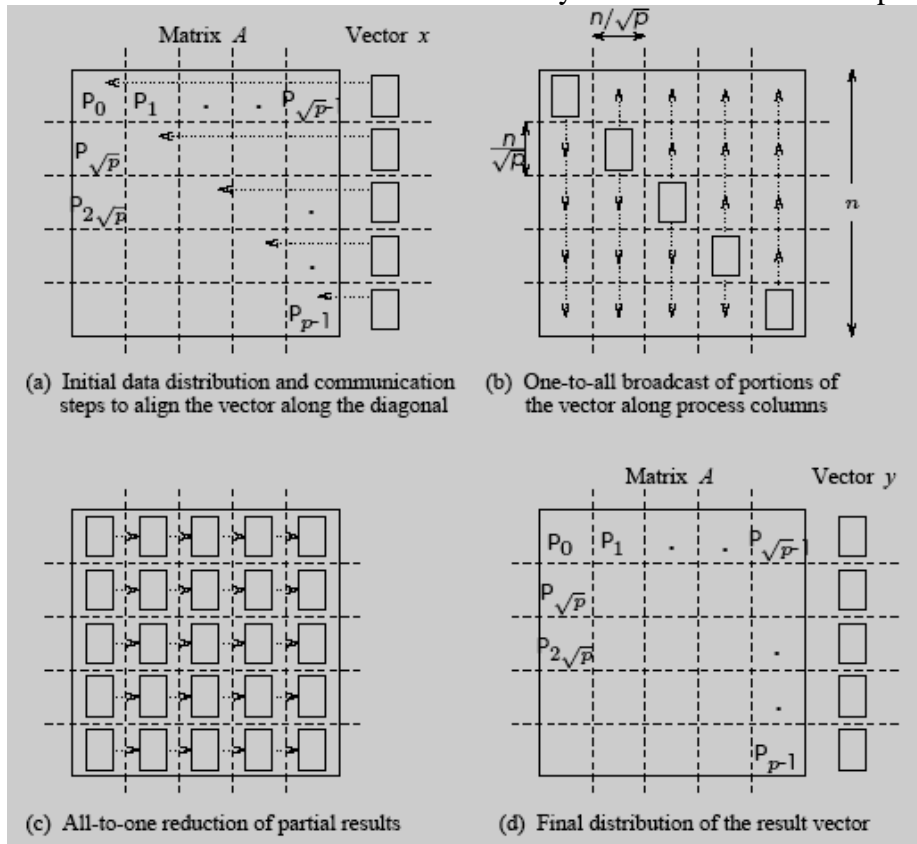
Multiplication of an  $n \times n$  matrix with an  $n \times 1$  vector using rowwise block 1-D partitioning. For the one-row-per-process case,  $p = n$ .

### Scalability Analysis:

- We know that  $T_o = pT_p - W$ , therefore, we have,
- $T_o = t_s p \log p + t_w np$
- For isoefficiency, we have  $W = K T_o^2$ , where  $K = E/(1 - E)$  for desired efficiency  $E$ .
- From this, we have  $W = O(p^2)$  (from the  $t_w$  term).
- There is also a bound on isoefficiency because of concurrency. In this case,  $p < n$ , therefore,  $W = n^2 = \Omega(p^2)$ .
- Overall isoefficiency is  $W = O(p^2)$ .

### Matrix-Vector Multiplication: 2-D Partitioning

- The  $n \times n$  matrix is partitioned among  $n$  processors such that each processor owns a single element.
- The  $n \times 1$  vector  $x$  is distributed only in the last column of  $n$  processors.



Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case,  $p = n$  if the matrix size is  $n \times n$ .

- We must first align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning aligns the vector  $x$  along the principal diagonal of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using  $n$  simultaneous broadcasts among all processors in the column.
- Finally, the result vector is computed by performing an all-to-one reduction along the columns.
  - Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the  $n$  processes of each column, and all-to-one reduction in each row.
  - Each of these operations takes  $\Theta(\log n)$  time and the parallel time is  $\Theta(\log n)$ .
- The cost (process-time product) is  $\Theta(n \log n)$ ; hence, the algorithm is not cost-optimal. When using fewer than  $n$  processors, each process owns an  $(n/\sqrt{p}) \times (n/\sqrt{p})$  block of the matrix.
  - The vector is distributed in portions of  $(n/\sqrt{p})$  elements in the last process-column only.

- In this case, the message sizes for the alignment, broadcast, and reduction are all  $(n/\sqrt{p})$ .

The computation is a product of an  $(n/\sqrt{p}) \times (n/\sqrt{p})$  submatrix with a vector of length  $(n/\sqrt{p})$

- The first alignment step takes time
- $t_s + t_w n / \sqrt{p}$
- The broadcast and reductions take time
- $t_s + t_w n / \sqrt{p} (\log \sqrt{p})$
- Local matrix-vector products take time  $t_c n^2 / p$
- Total time is  $T_p \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$
- Scalability Analysis:
- $T_o = p t_p - W = t_s p \log p + t_w n \sqrt{p} (\log p)$
- Equating  $T_o$  with  $W$ , term by term, for isoefficiency, we have,  $W = K^2 t_w^2 p \log^2 p$  as the dominant term.
- The isoefficiency due to concurrency is  $O(p)$ .
- The overall isoefficiency is  $p \log^2 p$  (due to the network bandwidth).
- For cost optimality, we have,  $W = n^2 = p \log^2 p$ . For this, we have,  $p = O\left(\frac{n^2}{\log^2 n}\right)$

### • **Matrix-Matrix Multiplication**

- Consider the problem of multiplying two  $n \times n$  dense, square matrices  $A$  and  $B$  to yield the product matrix  $C = A \times B$ .
- The serial complexity is  $O(n^3)$ .
- We do not consider better serial algorithms (Strassen's method), although, these can be used as serial kernels in the parallel algorithms.
- A useful concept in this case is called *block* operations. In this view, an  $n \times n$  matrix  $A$  can be regarded as a  $q \times q$  array of blocks  $A_{i,j}$  ( $0 \leq i, j < q$ ) such that each block is an  $(n/q) \times (n/q)$  submatrix.
- In this view, we perform  $q^3$  matrix multiplications, each involving  $(n/q) \times (n/q)$  matrices.
- Consider two  $n \times n$  matrices  $A$  and  $B$  partitioned into  $p$  blocks  $A_{i,j}$  and  $B_{i,j}$  ( $0 \leq i, j < \sqrt{p}$ ) of size  $(n/\sqrt{p}) \times (n/\sqrt{p})$  each.
- Process  $P_{i,j}$  initially stores  $A_{i,j}$  and  $B_{i,j}$  and computes block  $C_{i,j}$  of the result matrix.
- Computing submatrix  $C_{i,j}$  requires all submatrices  $A_{i,k}$  and  $B_{k,j}$  for  $0 \leq k < \sqrt{p}$ .
- All-to-all broadcast blocks of  $A$  along rows and  $B$  along columns.
- Perform local submatrix multiplication.
- The two broadcasts take time  $2t_s \log \sqrt{p} + t_w (n^2/p)(\sqrt{p} - 1)$

The computation requires  $\sqrt{p}$  multiplications of  $(n/\sqrt{p}) \times (n/\sqrt{p})$  sized submatrices.

- The parallel run time is approximately

$$T_p = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$$

- The algorithm is cost optimal and the isoefficiency is  $O(p^{1.5})$  due to bandwidth term  $t_w$  and concurrency.
- Major drawback of the algorithm is that it is not memory optimal.
- 

## • Matrix-Matrix Multiplication:

### Cannon's Algorithm

- In this algorithm, we schedule the computations of the  $\sqrt{p}$  processes of the  $i$ th row such that, at any given time, each process is using a different block  $A_{i,k}$ .
- These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh  $A_{i,k}$  after each rotation.

### Communication steps in Cannon's algorithm on 16 processes

- Align the blocks of  $A$  and  $B$  in such a way that each process multiplies its local submatrices. This is done by shifting all submatrices  $A_{i,j}$  to the left (with wraparound) by  $i$  steps and all submatrices  $B_{i,j}$  up (with wraparound) by  $j$  steps.
- Perform local block multiplication.
- Each block of  $A$  moves one step left and each block of  $B$  moves one step up (again with wraparound).
- Perform next block multiplication, add to partial result, repeat until all  $\sqrt{p}$  blocks have been multiplied.
- In the alignment step, since the maximum distance over which a block shifts is  $\sqrt{p} - 1$ , the two shift operations require a total of  $2(t_s + t_w(n^2 / p))$  time.
- Each of the  $\sqrt{p}$  single-step shifts in the compute-and-shift phase of the algorithm takes  $t_s + t_w(n^2 / p)$  time.
- The computation time for multiplying  $\sqrt{p}$  matrices of size  $(n / \sqrt{p}) \times (n / \sqrt{p})$  is  $n^3 / p$
- The parallel time is approximately:  $T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}$
- The cost-efficiency and isoefficiency of the algorithm are identical to the first algorithm, except, this is memory optimal.
-

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

(a) Initial alignment of A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

(b) Initial alignment of B

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{2,2}$	$A_{2,3}$	$A_{2,0}$	$A_{2,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{3,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$

(c) A and B after initial alignment

$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{1,2}$	$A_{1,3}$	$A_{1,0}$	$A_{1,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{2,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$

(d) Submatrix locations after first shift

$A_{0,2}$	$A_{0,3}$	$A_{0,0}$	$A_{0,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$
$A_{1,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$

(e) Submatrix locations after second shift

$A_{0,3}$	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$
$B_{3,0}$	$B_{0,1}$	$B_{1,2}$	$B_{2,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$B_{0,0}$	$B_{1,1}$	$B_{2,2}$	$B_{3,3}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,0}$
$B_{1,0}$	$B_{2,1}$	$B_{3,2}$	$B_{0,3}$
$A_{3,2}$	$A_{3,3}$	$A_{3,0}$	$A_{3,1}$
$B_{2,0}$	$B_{3,1}$	$B_{0,2}$	$B_{1,3}$

(f) Submatrix locations after third shift

## Programming Using the Message-Passing Paradigm

Numerous programming languages and libraries have been developed for explicit parallel programming. These differ in their view of the address space that they make available to the programmer, the degree of synchronization imposed on concurrent activities, and the multiplicity of programs. The message-passing programming paradigm is one of the oldest and most widely used approaches for programming parallel computers. Its roots can be traced back in the early days of parallel processing and its wide-spread adoption can be attributed to the fact that it imposes minimal requirements on the underlying hardware.



## Principles of Message-Passing Programming

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelization.

The logical view of a machine supporting the message-passing paradigm consists of  $p$  processes, each with its own exclusive address space. Instances of such a view come naturally from clustered workstations and non-shared address space multicomputers. There are two immediate implications of a partitioned address space. First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access that is critical for achieving high performance on non-UMA architectures, since a processor can access its local data much faster than non-local data on such architectures. The second implication is that all interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data. This requirement for cooperation adds a great deal of complexity for a number of reasons. The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason. However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions. Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.

The message-passing programming paradigm requires that the parallelism is coded explicitly by the programmer. That is, the programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency. As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

### Structure of Message-Passing Programs

Message-passing programs are often written using the asynchronous or loosely synchronous paradigms. In the asynchronous paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have non-deterministic behavior due to race conditions. Loosely synchronous programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.

In its most general form, the message-passing paradigm supports execution of a different program on each of the  $p$  processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable. For this reason, most message-passing programs are written using the single program multiple data (SPMD) approach. In SPMD programs

the code executed by different processes is identical except for a small number of processes (e.g., the "root" process). This does not mean that the processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

### The Building Blocks: Send and Receive Operations

Since interactions are accomplished by sending and receiving messages, the basic operations in the message-passing programming paradigm are `send` and `receive`. In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

The `sendbuf` points to a buffer that stores the data to be sent, `recvbuf` points to a buffer that stores the data to be received, `nelems` is the number of data units to be sent and received, `dest` is the identifier of the process that receives the data, and `source` is the identifier of the process that sends the data.

However, to stop at this point would be grossly simplifying the programming and performance ramifications of how these functions are implemented. To motivate the need for further investigation, let us start with a simple example of a process sending a piece of data to another process as illustrated in the following code-fragment:

```
1           P0                               P1
2
3           a = 100;
4           send(&a, 1, 1);
5           a=0;
           receive(&a, 1, 0)
           printf("%d\n", a);
```

In this simple example, process P0 sends a message to process P1 which receives and prints the message. The important thing to note is that process P0 changes the value of `a` to 0 immediately following the `send`. The semantics of the `send` operation require that the value received by process P1 must be 100 as opposed to 0. That is, the value of `a` at the time of the `send` operation must be the value that is received by process P1.

It may seem that it is quite straightforward to ensure the semantics of the `send` and `receive` operations. However, based on how the `send` and `receive` operations are implemented this may not be the case. Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware. Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed). As a result, if the `send` operation programs the communication hardware and returns before the communication operation has been accomplished, process P1 might receive the value 0 in `a` instead of 100!



While this is undesirable, there are in fact reasons for supporting such send operations for performance reasons. In the rest of this section, we will discuss send and receive operations in the context of such a hardware environment, and motivate various implementation details and message-passing protocols that help in ensuring the semantics of the send and receive operations.

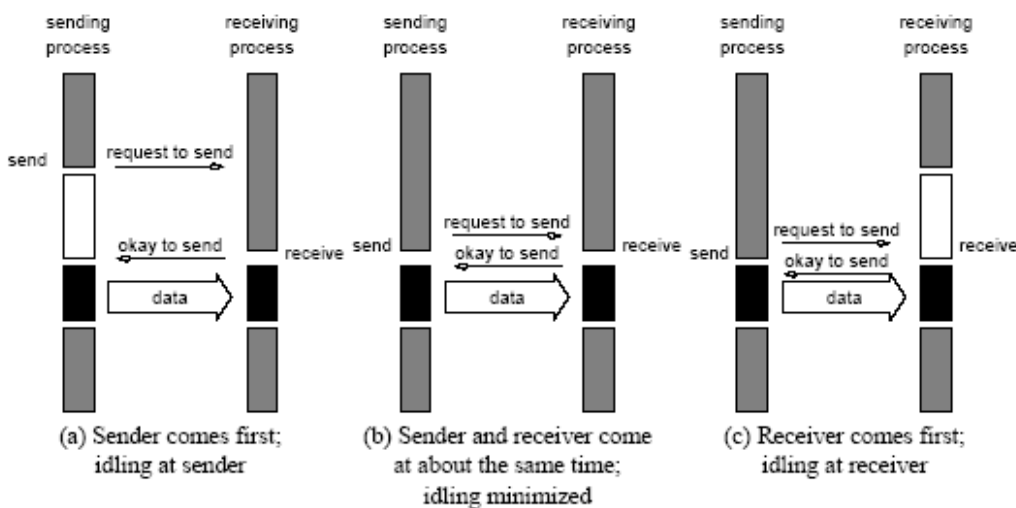
## 1 Blocking Message Passing Operations

A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so. Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanisms by which this can be achieved.

### Blocking Non-Buffered Send/Receive

In the first case, the send operation does not return until the matching receive has been encountered at the receiving process. When this happens, the message is sent and the send operation returns upon completion of the communication operation. Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation. The operation is illustrated in the figure. Since there are no buffers used at either sending or receiving ends, this is also referred to as a non-buffered blocking operation.

**Figure 1. Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.**



**Idling Overheads in Blocking Non-Buffered Operations** In figure-1, we illustrate three scenarios in which the send is reached before the receive is posted, the send and receive are posted around the same time, and the receive is posted before the send is reached. In cases (a) and (c), we notice that there is considerable idling at the sending and receiving process. It is also clear from the figures that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same

time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.

**Deadlocks in Blocking Non-Buffered Operations** Consider the following simple exchange of messages that can lead to a deadlock:

1	P0	P1
2		
3	<code>send(&amp;a, 1, 1);</code>	<code>send(&amp;a, 1, 0);</code>
4	<code>receive(&amp;b, 1, 1);</code>	<code>receive(&amp;b, 1, 0);</code>

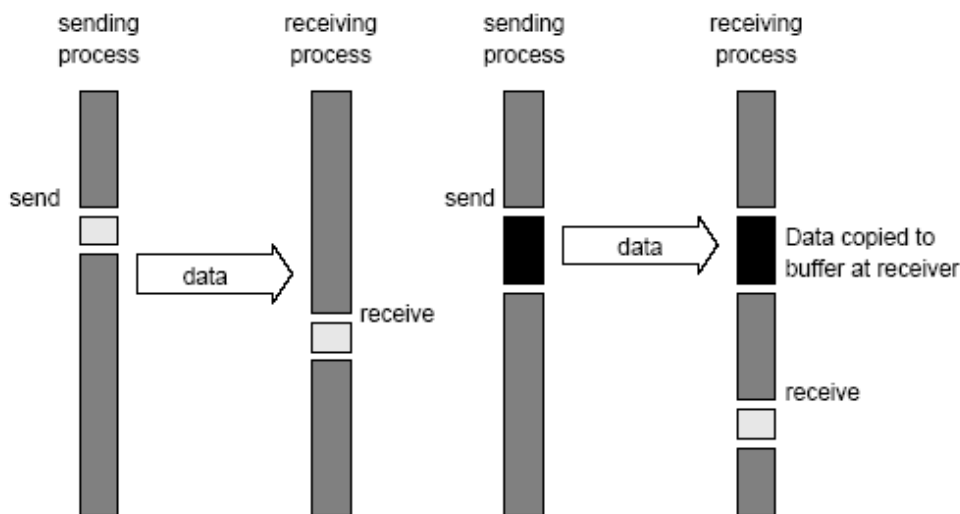
The code fragment makes the values of `a` available to both processes P0 and P1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P0 waits for the matching receive at P1 whereas the send at process P1 waits for the corresponding receive at P0, resulting in an infinite wait.

As can be inferred, deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined. In the above example, this can be corrected by replacing the operation sequence of one of the processes by a `receive` and a `send` as opposed to the other way around. This often makes the code more cumbersome and buggy.

### Blocking Buffered Send/Receive

A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends. We start with a simple case in which the sender has a buffer pre-allocated for communicating messages. On encountering a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The sender process can now continue with the program knowing that any changes to the data will not impact program semantics. The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer. Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well. When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location. This operation is illustrated in Figure 2(a).

**Figure 2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.**



In the protocol illustrated above, buffers are used at both sender and receiver and communication is handled by dedicated hardware. Sometimes machines do not have such communication hardware. In this case, some of the overhead can be saved by buffering only on one side. For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end. When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location. This protocol is illustrated in Figure 2(b). It is not difficult to conceive a protocol in which the buffering is done only at the sender and the receiver initiates a transfer by interrupting the sender.

It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads. In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

### Example 1 Impact of finite buffers in message passing

Consider the following code fragment:

```

1          P0                                P1
2
3          for (i = 0; i < 1000; i++) {      for (i = 0; i < 1000; i++) {
4              produce_data(&a);              receive(&a, 1, 0);
5              send(&a, 1, 1);                consume_data(&a);
6          }                                  }

```

In this code fragment, process P0 produces 1000 data items and process P1 consumes them. However, if process P1 was slow getting to this loop, process P0 might have sent all of its data. If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space. This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements.

Deadlocks in Buffered Send and Receive Operations While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

1	P0	P1
2		
3	<code>receive(&amp;a, 1, 1);</code>	<code>receive(&amp;a, 1, 0);</code>
4	<code>send(&amp;b, 1, 1);</code>	<code>send(&amp;b, 1, 0);</code>

Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

## 2 Non-Blocking Message Passing Operations

In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered). Often, it is possible to require the programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead. This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation. Non-blocking operations are generally accompanied by a `check-status` operation, which indicates whether the semantics of a previously initiated transfer may be violated or not. Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation. Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.

As illustrated in figure 3, non-blocking operations can themselves be buffered or non-buffered. In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated. When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data. This transfer is indicated in figure 4(a).

Figure 3. Space of possible protocols for send and receive operations.

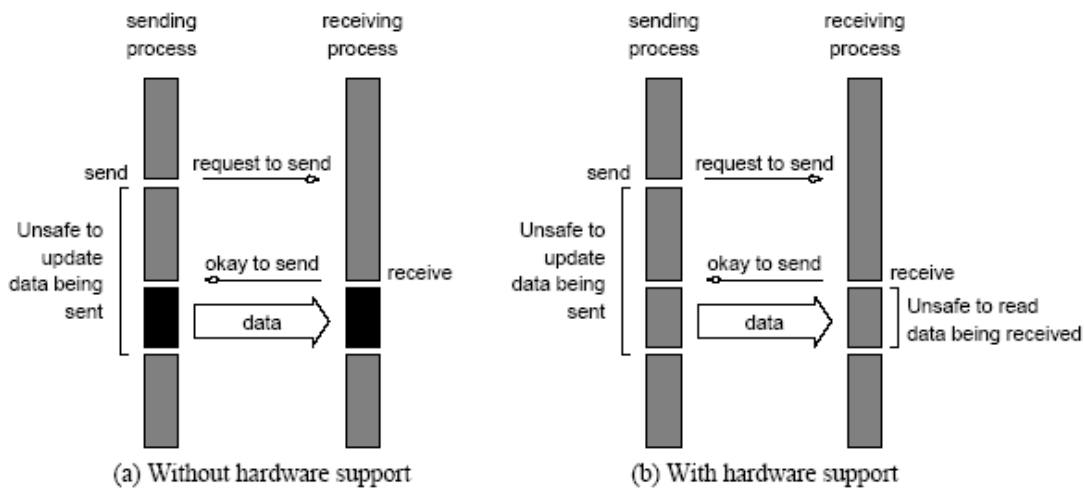
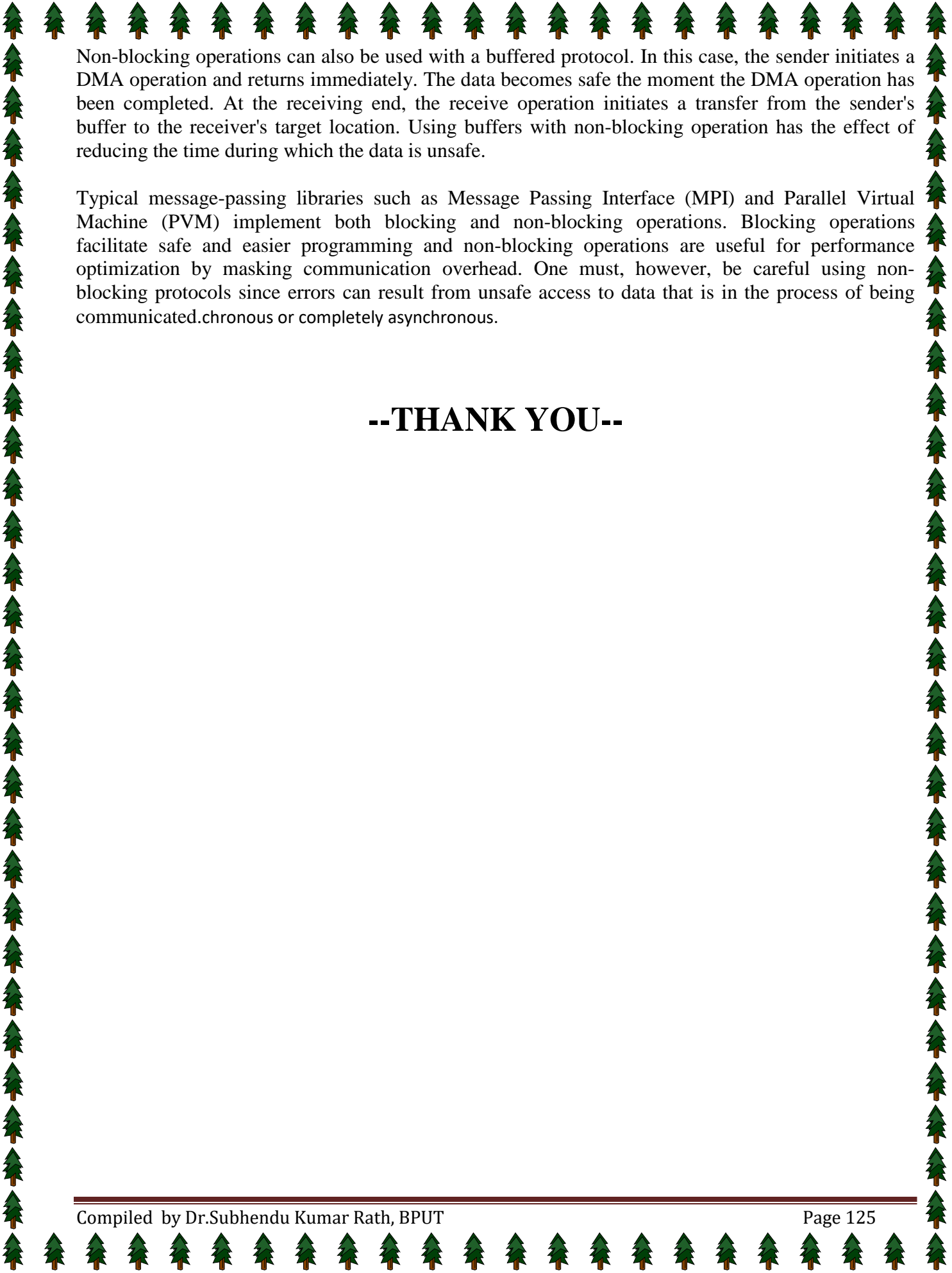


Figure 4. Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Non-Buffered	Sending process blocks until matching receive operation has been encountered	
	Send and Receive semantics assured by corresponding operation	Programmer must explicitly ensure semantics by polling to verify completion

Comparing Figures 4(a) and 1(a), it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent. This alleviates the major bottleneck associated with the former at the expense of some program restructuring. The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware. This is illustrated in figure 4(b). In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case, however, the data being received is unsafe for the duration of the receive operation.



Non-blocking operations can also be used with a buffered protocol. In this case, the sender initiates a DMA operation and returns immediately. The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location. Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations. Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead. One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.chronous or completely asynchronous.

**--THANK YOU--**