DIGITAL NOTES ON
Parallel and Distributed Computing
(R18A0530)

B.TECH IV YEAR - II SEM
(2022-23)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – ‘A’ Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

Professional Elective-IV (R18A0530) Parallel and Distributed Computing

Course Objective: To learn the advanced concepts of Parallel and Distributed Computing and its implementation for assessment of understanding the course by the students

UNIT-I

Introduction: Scope, issues, applications and challenges of Parallel and Distributed Computing **Parallel Programming Platforms**: Implicit Parallelism: Trends in Microprocessor Architectures,Dichotomy of Parallel Computing Platforms, Physical Organization, co-processing.

UNIT-II

Principles of Parallel Algorithm Design: Decomposition Techniques, Characteristics of Tasks andInteractions, Mapping Techniques for Load Balancing.

CUDA programming model: Overview of CUDA, Isolating data to be used by parallelized code,API function to allocate memory on parallel computing device, to transfer data.

UNIT-III

Analytical Modeling of Parallel Programs: Sources of Overhead in Parallel Programs, Performance Metrics for Parallel Systems, The Effect of Granularity on Performance, Scalability of Parallel Systems, Minimum Execution Time and Minimum Cost Optimal Execution Time

UNIT-IV

Dense Matrix Algorithms: Matrix-Vector Multiplication, Matrix-Matrix Multiplication, Issues in Sorting on Parallel Computers, Bubble Sort and Variants, Quick Sort Algorithm.

UNIT-V

Search Algorithms for Discrete Optimization Problems: Sequential Search Algorithms, ParallelDepth-First Search, Parallel Best-First Search, Speed up Anomalies in Parallel Search Algorithms

Course outcomes

1. Gain basic understanding of fundamental concepts in parallel computing.
2. Be able to identify and leverage common parallel computing patterns.
3. Be able to properly assess efficiency and scalability of a parallel algorithm/application.
4. Become proficient in using at least one parallel programming technique, and familiar with several others.

Recommended Books:

1. A Grama, AGupra, G Karypis, V Kumar. Introduction to Parallel Computing (2nd ed.). Addison Wesley,2003.

2. C Lin, L Snyder. Principles of Parallel Programming. USA: Addison-Wesley PublishingCompany,2008. 3. J Jeffers, J Reinders. Intel Xeon Phi Coprocessor High-Performance Programming.Morgan Kaufmann Publishing and Elsevier,2013.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF CSE

INDEX

Professional Elective-IV
(R18A0530) Parallel and Distributed Computing

UNIT-I

Introduction: Scope, issues, applications and challenges of Parallel and Distributed Computing

Parallel Programming Platforms: Implicit Parallelism: Trends in Microprocessor Architectures, Dichotomy of Parallel Computing Platforms, Physical Organization, co-processing.

Parallel Computing:

Parallel computing refers to the process of executing several processors an application or computation simultaneously. Generally, it is a kind of computing architecture where the large problems break into independent, smaller, usually similar parts that can be processed in one go. It is done by multiple CPUs communicating via shared memory, which combines results upon completion. It helps in performing large computations as it divides the large problem between more than one processor.

Distributed Computing

In distributed computing we have multiple autonomous computers which seems to the user as single system. In distributed systems there is no shared memory and computers communicate with each other through message passing. In distributed computing a single task is divided among different computers.

Difference between Parallel Computing and Distributed Computing:

S.No.	**Parallel Computing**	**Distributed Computing**
1	Many operations are performed simultaneously	System components are located at different locations
2	Single computer is required	Uses multiple computers
3	Multiple processors perform multiple operations	Multiple computers perform multiple operations
4	It may have shared or distributed memory	It have only distributed memory
5	Processors communicate with each other through bus	Computer communicate with each other through message passing.
6	Improves the system performance	Improves system scalability, fault tolerance and resource sharing capabilities

Types of parallel computing

From the open-source and proprietary parallel computing vendors, there are generally three types of parallel computing available, which are discussed below:

1. Bit-level parallelism: The form of parallel computing in which every task is dependent on processor size. In terms of performing a task on large-sized data, it reduces the number of instructions the processor must execute. There is a need to split the operation into series of instructions. For example, there is an 8-bitprocessor, and you want to do an operation on 16-bit numbers. First, it must operate the 8 lower-order bits and then the 8 higher-order bits. Therefore, two instructions are needed to execute the operation. Theoperation can be performed with one instruction by a 16-bit processor.
2. Instruction-level parallelism: In a single CPU clock cycle, the processor decides in instruction-level parallelism how many instructions are implemented at the same time. For each clock cycle phase, a processor in instruction-level parallelism can have the ability to address that is less than one instruction. Thesoftware approach in instruction-level parallelism functions on static parallelism, where the computer decideswhich instructions to execute simultaneously.
3. Task Parallelism: Task parallelism is the form of parallelism in which the tasks are decomposed into subtasks. Then, each subtask is allocated for execution. And, the execution of subtasks is performed concurrently by processors.

Applications of Parallel Computing

There are various applications of Parallel Computing, which are as follows:

- One of the primary applications of parallel computing is Databases and Data mining.
- The real-time simulation of systems is another use of parallel computing.
- The technologies, such as Networked videos and Multimedia.
- Science and Engineering.
- Collaborative work environments.
- The concept of parallel computing is used by augmented reality, advanced graphics, and virtual reality.

Advantages of Parallel computing

Parallel computing advantages are discussed below:

- In parallel computing, more resources are used to complete the task that led to decrease the time and cut possible costs. Also, cheap components are used to construct parallel clusters.
- Comparing with Serial Computing, parallel computing can solve larger problems in a short time.
- For simulating, modeling, and understanding complex, real-world phenomena, parallel computing is much appropriate while comparing with serial computing.
- When the local resources are finite, it can offer benefit you over non-local resources.
- There are multiple problems that are very large and may impractical or impossible to solve them on a single computer; the concept of parallel computing helps to remove these kinds of issues.
- One of the best advantages of parallel computing is that it allows you to do several things in a time by using multiple computing resources.
- Furthermore, parallel computing is suited for hardware as serial computing wastes the potential computing power.

Disadvantages of Parallel Computing

There are many limitations of parallel computing, which are as follows:

- It addresses Parallel architecture that can be difficult to achieve.
- In the case of clusters, better cooling technologies are needed in parallel computing.
- It requires the managed algorithms, which could be handled in the parallel mechanism.
- The multi-core architectures consume high power consumption.
- The parallel computing system needs low coupling and high cohesion, which is difficult to create.
- The code for a parallelism-based program can be done by the most technically skilled and expert programmers.
- Although parallel computing helps you out to resolve computationally and the data-exhaustive issue with the help of using multiple processors, sometimes it affects the conjunction of the system and some of our control algorithms and does not provide good outcomes due to the parallel option.
- Due to synchronization, thread creation, data transfers, and more, the extra cost sometimes can be quite large; even it may be exceeding the gains because of parallelization.
- Moreover, for improving performance, the parallel computing system needs different code tweaking for different target architectures.

Future of Parallel Computing

From serial computing to parallel computing, the computational graph has completely changed. Tech giant likes Intel has already started to include multicore processors with systems, which is a great step towards parallel computing. For a better future, parallel computation will bring a revolution in the way of working the computer. Parallel Computing plays an important role in connecting the world with each other more than before. Moreover, parallel computing's approach becomes more necessary with multi-processor computers, faster networks, and distributed systems.

Parallel and distributed computing occurs across many different topic areas in computer science, including algorithms, computer architecture, networks, operating systems, and software engineering.

Platform-based development

Platform-based development is concerned with the design and development of applications for specific types of computers and operating systems (“platforms”). Platform-based development takes into account system-specific characteristics, such as those found in Web programming, multimedia development, mobile application development, and robotics.

Security and information assurance

Security and information assurance refers to policy and technical elements that protect information systems by ensuring their availability, integrity, authentication, and appropriate levels of confidentiality. Information security concepts occur in many areas of computer science, including operating systems, computer networks, databases, and software.

Software engineering

Software engineering is the discipline concerned with the application of theory, knowledge, and practice to building reliable software systems that satisfy the computing requirements of customers and users. It is applicable to small-, medium-, and large-scale computing systems and organizations. Software engineering uses engineering methods, processes, techniques, and measurements. Software development, whether done by an individual or a team, requires choosing the most appropriate tools, methods, and approaches for a given environment.

Social and professional issues

Computer scientists must understand the relevant social, ethical, and professional issues that surround their activities. The ACM Code of Ethics and Professional Conduct provides a basis for personal responsibility and professional conduct for computer scientists who are engaged in system development that directly affects the general public.

Issues present in parallel and distributed paradigms:

Traditionally, distributed computing focused on resource availability, result correctness, code portability and transparency of access to the resources more than on issues of efficiency and speed which, in

addition to scalability, are central to parallel computing.
What are the issues in distributed computing?

Issues in Distributed Systems

- The lack of global knowledge.
- Naming.
- Scalability.
- Compatibility.
- Process synchronization (requires global knowledge)
- Resource management (requires global knowledge)
- Security.
- Fault tolerance, error recovery.

Parallel computers can be classified according to the level at which the architecture supports parallelism, with multi-core and multi-processor computers The paper proceeds by specifying key design issues of operating system: like processes synchronization, memory management, communication, concurrency control.

Applications of parallel and distributed computing:

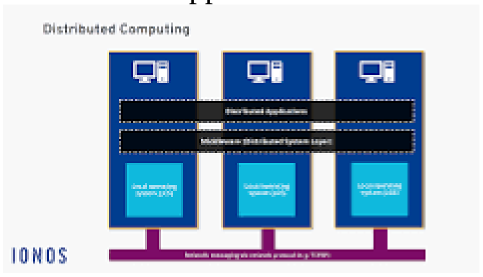
Applications of parallel computing:

Notable applications for parallel processing (also known as parallel computing) include

(1) computational astrophysics
(2) geo processing (or seismic surveying)
(3) climate modeling
(4) agriculture estimates
(5) financial risk management
(6) video color correction
(7) computational fluid dynamics
(8) medical imaging and drug discovery

Applications of Distributed computing:

What are the applications of distributed computing?



Social networks, mobile systems, online banking, and online gaming (e.g. multiplayer systems) also use efficient distributed systems. Additional areas of application for distributed computing include e-learning platforms, artificial intelligence, and e-commerce

Challenges of Parallel and distributed Systems:

What are the challenges of parallel and distributed computing?
Important concerns are workload sharing, which attempts to take advantage of access to multiple computers tc complete jobs faster; task migration, which supports workload sharing by efficiently distributing jobs among machines; and automatic task replication, which occurs at different sites for greater reliability.
Challenges of distributed Systems

(1) Heterogeneity: The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. ...
(2) Transparency: ...
(3) Openness. ...
(4) Concurrency. ...
(5) Security. ...
(6) Scalability. ...
(7) Failure Handling.

Parallel Programming Platforms: Implicit Parallelism: Trends in Microprocessor Architectures, Dichotomy o Parallel Computing Platforms, Physical Organization, co-processing.

Scope of Parallelism

• Conventional architectures coarsely comprise of a processor, memory system, and the data path.

• Each of these components present significant performance bottlenecks.

• Parallelism addresses each of these components in significant ways.

• Different applications utilize different aspects of parallelism – e.g., data intensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.

• It is important to understand each of these performance bottlenecks.

Implicit Parallelism: Trends in Microprocessor Architectures

Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude)

Higher levels of device integration have made available a large number of transistors.

- The question of how best to utilize these resources is an important one.
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

Pipelining and Superscalar Execution

• Pipelining overlaps various stages of instruction execution to achieve performance.

• At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.

• This is akin to an assembly line for manufacture of cars

• Pipelining, however, has several limitations.

• The speed of a pipeline is eventually limited by the slowest stage.

• For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).

• However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.

- The penalty of a mis prediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.
- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.

Superscalar Execution Scheduling of instructions is determined by a number of factors:

• True Data Dependency: The result of one operation is an input to the next.

• Resource Dependency: Two operations require the same resource.

• Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.

• The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.

• The complexity of this hardware is an important constraint on superscalar processors.

Very Long Instruction Word (VLIW) Processors

• The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.

• To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.

• These instructions are packed and dispatched together, and thus the name very long instruction word.

• This concept was used with some commercial success in the Multiflow Trace machine (circa 1984).

• Variants of this concept are employed in the Intel IA64 processors.

Very Long Instruction Word (VLIW) Processors: Considerations

• Issue hardware is simpler.

• Compiler has a bigger context from which to select coscheduled instructions.

• Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.

• Branch and memory prediction is more difficult.

• VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.

• Typical VLIW processors are limited to 4-way to 8-way parallelism.

Dichotomy of Parallel Computing Platforms:

- An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
- The former is sometimes also referred to as the control structure and the latter as the communication model.

Control Structure of Parallel Programs

• Parallelism can be expressed at various levels of granularity – from instruction level to processes.

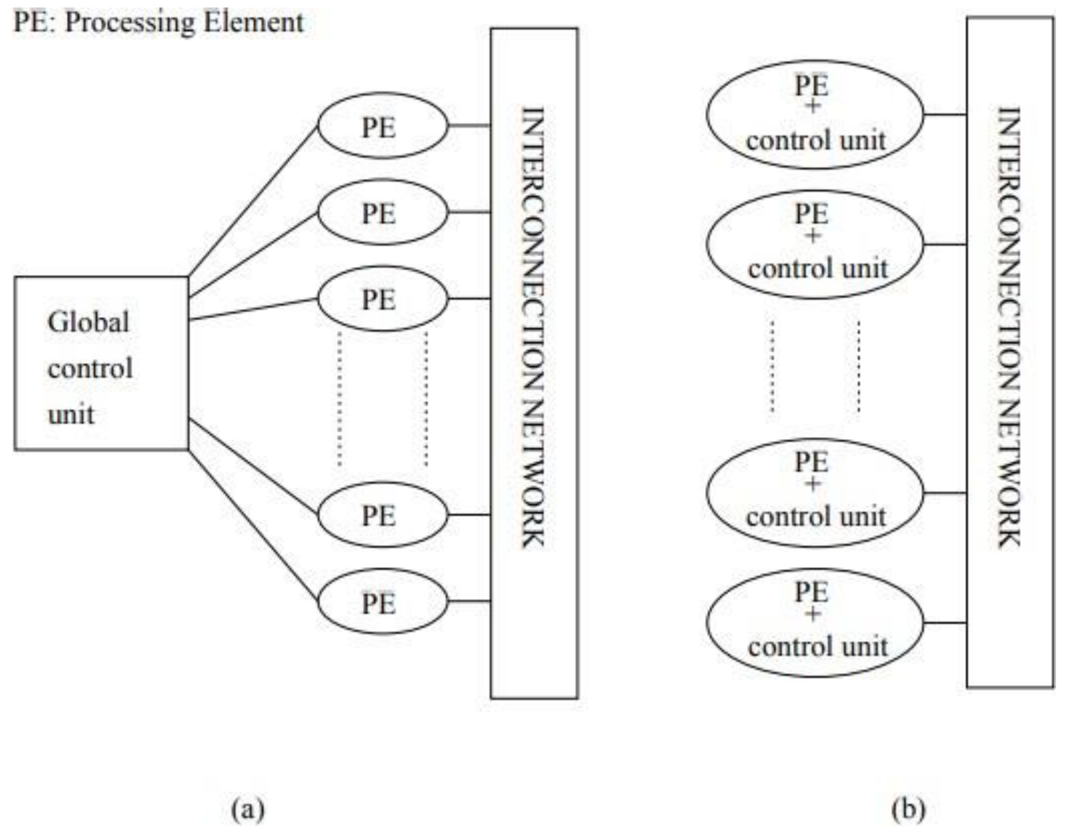
• Between these extremes exist a range of models, along with corresponding architectural support.

• Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.

• If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).

• If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

SIMD and MIMD Processors



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

SIMD Processors

• Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines.

• Variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc.

• SIMD relies on the regular structure of computations (such as those in image processing).

• It is often necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an “activity mask”, which determines if a processor should participate in a computation or not.

MIMD Processors

• In contrast to SIMD processors, MIMD processors can execute different programs on different processors.

• A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.

• It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.

• Examples of such platforms include current generation Sun Ultra Servers, SGI Origin Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD-MIMD Comparison

• SIMD computers require less hardware than MIMD computers (single control unit).

• However, since SIMD processors ae specially designed, they tend to be expensive and have long design cycles.

• Not all applications are naturally suited to SIMD processors.

• 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.

Communication Model of Parallel Platforms

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

• Platforms that provide a shared data space are called shared address-space machines or multiprocessors.

• Platforms that support messaging are also called message passing platforms or multi computers.

- **Shared-Address-Space Platforms**

• Part (or all) of the memory is accessible to all processors.

• Processors interact by modifying data objects stored in this shared-address-space.

• 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), else, a non uniform memory access (NUMA) machine.

- **NUMA and UMA Shared-Address-Space Platforms:**

• The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance. • Programming these platforms is easier since reads and writes are implicitly visible to other processors.

• However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).

• Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.

• A weaker model of these machines provides an address map, but not coordinated access. These models are called non cache coherent shared address space machines.

➢ **Shared-Address-Space vs. Shared Memory Machines**

• It is important to note the difference between the terms shared address space and shared memory.

• We refer to the former as a programming abstraction and to the latter as a physical machine attribute.

• It is possible to provide a shared address space using a physically distributed memory.

Physical Organization of Parallel Platforms:

We begin this discussion with an ideal parallel machine called Parallel Random Access Machine, or PRAM.

Architecture of an Ideal Parallel Computer

• A natural extension of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine, or PRAM.

• PRAMs consist of p processors and a global memory of unbounded size that is uniformly accessible to all processors.

• Processors share a common clock but may execute different instructions in each cycle.

Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

• Exclusive-read, exclusive-write (EREW) PRAM.

• Concurrent-read, exclusive-write (CREW) PRAM

• Exclusive-read, concurrent-write (ERCW) PRAM.

• Concurrent-read, concurrent-write (CRCW) PRAM.

• Common: write only if all values are identical.

• Arbitrary: write the data from a randomly selected processor.

• Priority: follow a predetermined priority order.

• Sum: Write the sum of all data items.

Physical Complexity of an Ideal Parallel Computer

• Processors and memories are connected via switches.

• Since these switches must operate in O(1) time at the level of words, for a system of p processors and m words, the switch complexity is O (mp).

• Clearly, for meaningful values of p and m, a true PRAM is not realizable.

UNIT-II

Principles of Parallel Algorithm Design: Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing.

CUDA programming model: Overview of CUDA, isolating data to be used by parallelized code, API function to allocate memory on parallel computing device, to transfer data.

Principles of Parallel Algorithm Design: Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing.

Decomposition Techniques:

Exploratory Decomposition:

Decomposition is fixed/static from the design

– Data and recursive

• Exploration (search) of a state space of solutions

– Problem decomposition reflects shape of execution

– Goes hand-in-hand with its execution

• Examples

– discrete optimization, e.g. 0/1 integer programming

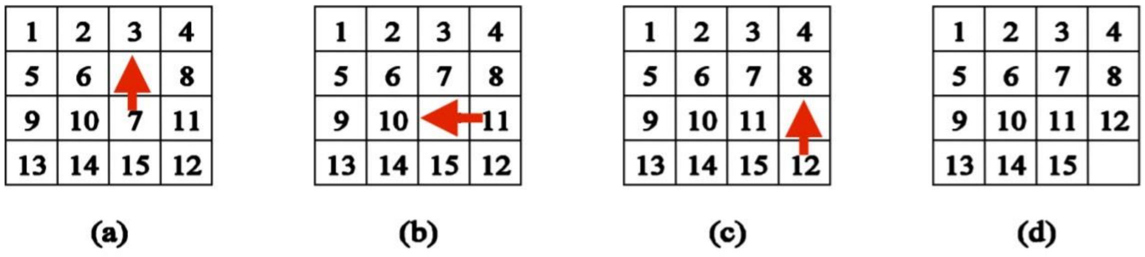
– theorem proving

– game playing

Example:

Solve a 15 puzzle

• Sequence of three moves from state (a) to final state (d)



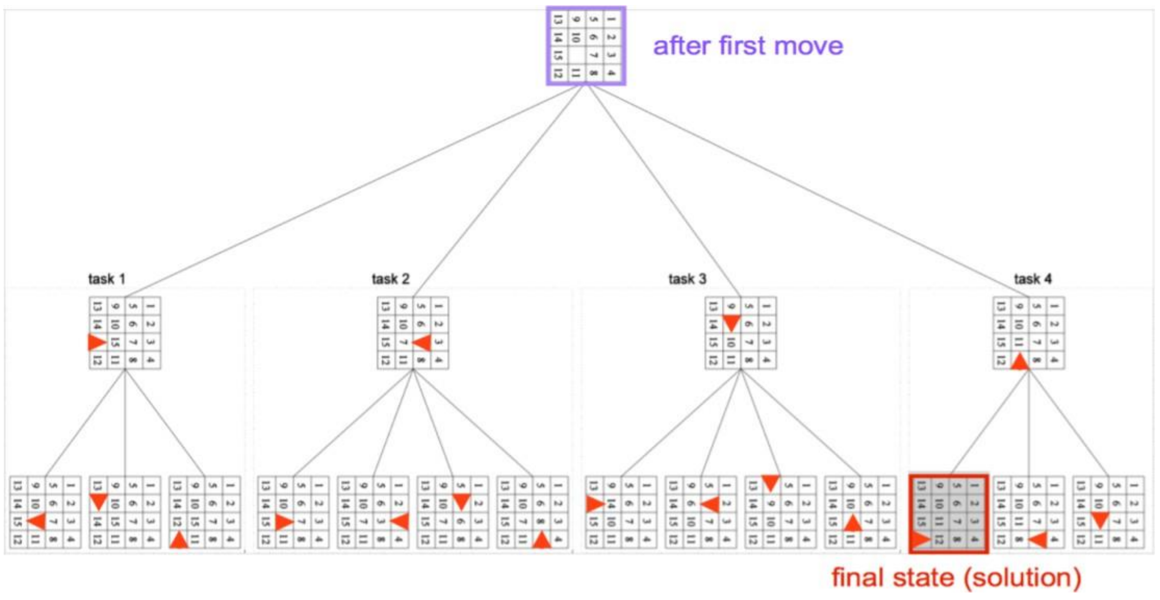
• From an arbitrary state, must search for a solution

Solving a 15 puzzle

• Search

– generate successor states of the current state

– explore each as an independent task

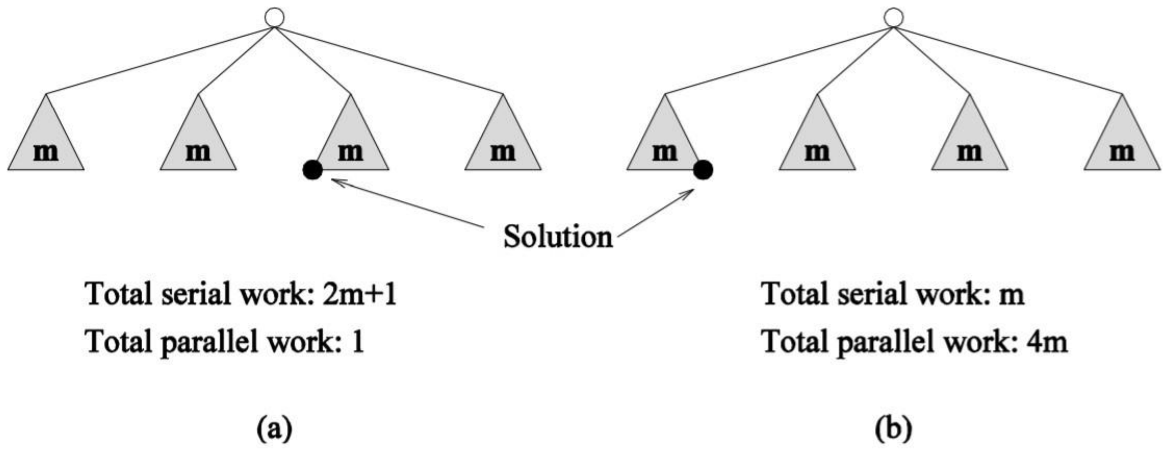


Exploratory Decomposition Speedup:

Solve a 15 puzzle

• The decomposition behaves according to the parallel formulation

– May change the amount of work done



Execution terminate when a solution is found

Speculative Decomposition

Dependencies between tasks are not known a-priori.

Impossible to identify independent tasks

• Two approaches

– Conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies

• May yield little concurrency

– Optimistic approaches, which schedule tasks even when they may potentially be inter-dependent

• Roll-back changes in case of an error

Discrete event simulation

• Centralized time-ordered event list

– you get up ->get ready->drive to work->work->eat lunchà->work some more->drive back->eat dinner->and sleep

- Simulation
  - – extract next event in time order

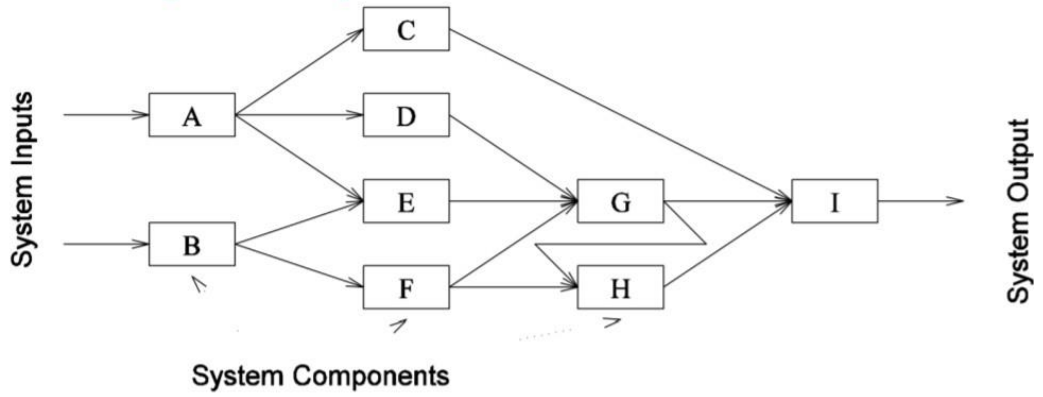
– process the event
– if required, insert new events into the event list

• Optimistic event scheduling
  – assume outcomes of all prior events
  – speculatively process next event
  – if assumption is incorrect, roll back its effects and continue

Simulation of a network of nodes

• Simulate network behavior for various input and node delays
  – The input are dynamically changing

• Thus task dependency is unknown



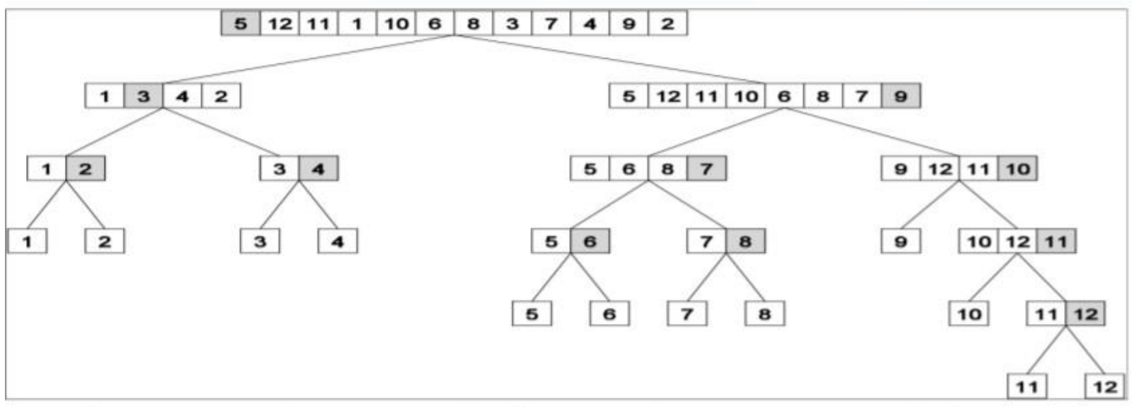
Speculate execution: tasks' input
  – Correct: parallelism
  – Incorrect: rollback and redo

Speculative vs Exploratory:

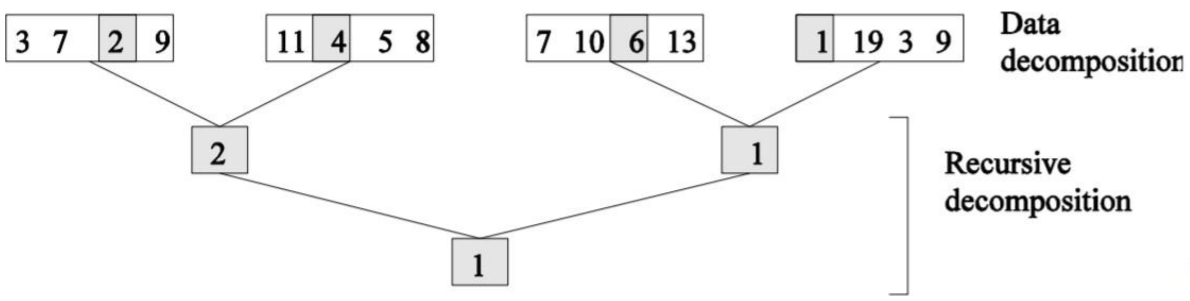
Exploratory decomposition
  – The output of multiple tasks from a branch is unknown
  – Parallel program perform more, less or same amount of work as serial program

• Speculative
  – The input at a branch leading to multiple parallel tasks is unknown
  – Parallel program perform more or same amount of work as the serial algorithm

Use multiple decomposition techniques together
• One decomposition may be not optimal for concurrency
  – Quicksort recursive decomposition limits concurrency (Why?)



Combined recursive and data decomposition for MIN



Characteristics of Tasks

Theory

– Decomposition: to parallelize theoretically

• Concurrency available in a problem

• Practice

– Task creations, interactions and mapping to PEs.

• Realizing concurrency practically

– Characteristics of tasks and task interactions

• Impact choice and performance of parallelism

• Characteristics of tasks

– Task generation strategies

– Task sizes (the amount of work, e.g. FLOPs)

– Size of data associated with tasks

Task Generation:

Static task generation

– Concurrent tasks and task graph known a-priori (before execution)

– Typically using recursive or data decomposition

– Examples

• Matrix operations

• Graph algorithms

• Image processing applications

• Other regularly structured problems

• **Dynamic task generation**

– Computations formulate concurrent tasks and task graph on the fly

• Not explicit a priori, though high-level rules or guidelines known

– Typically by exploratory or speculative decompositions.

• Also possible by recursive decomposition, e.g. quicksort

– A classic example: game playing

• 15 puzzle board

Task Sizes/Granularity

The amount of work à amount of time to complete

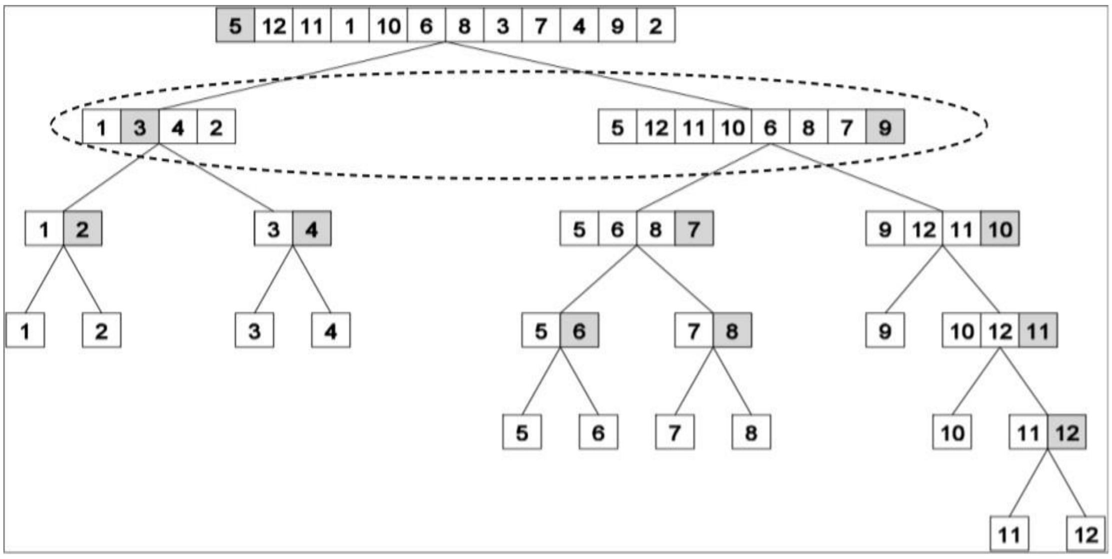
– E.g. FLOPs, #memory access

• Uniform:

– Often by even data decomposition, i.e. regular

• Non-uniform

– Quicksort, the choice of pivot



Size of Data Associated with Tasks:

May be small or large compared to the task sizes

– How relevant to the input and/or output data sizes

– Example:

• size(input) < size(computation), e.g., 15 puzzle

• size(input) = size(computation) > size(output), e.g., min

• size(input) = size(output) < size(computation), e.g., sort

• Considering the efforts to reconstruct the same task context

– small data: small efforts: task can easily migrate to another process

– large data: large efforts: ties the task to a process

• Context reconstructing vs communicating

– It depends

Characteristics of Task Interactions:

Aspects of interactions

– What: shared data or synchronizations, and sizes of the media

– When: the timing

– Who: with which task(s), and overall topology/patterns

– Do we know details of the above three before execution

– How: involve one or both?

• The implementation concern, implicit or explicit

Orthogonal classification

• Static vs. dynamic

• Regular vs. irregular

• Read-only vs. read-write

• One-sided vs. two-sided

• Aspects of interactions

– What: shared data or synchronizations, and sizes of the media

– When: the timing

– Who: with which task(s), and overall topology/patterns

– Do we know details of the above three before execution

– How: involve one or both?

• Static interactions

– Partners and timing (and else) are known a-priori

– Relatively simpler to code into programs.

- Dynamic interactions
  - The timing or interacting tasks cannot be determined a-priori.

– Harder to code, especially using explicit interaction.

Aspects of interactions

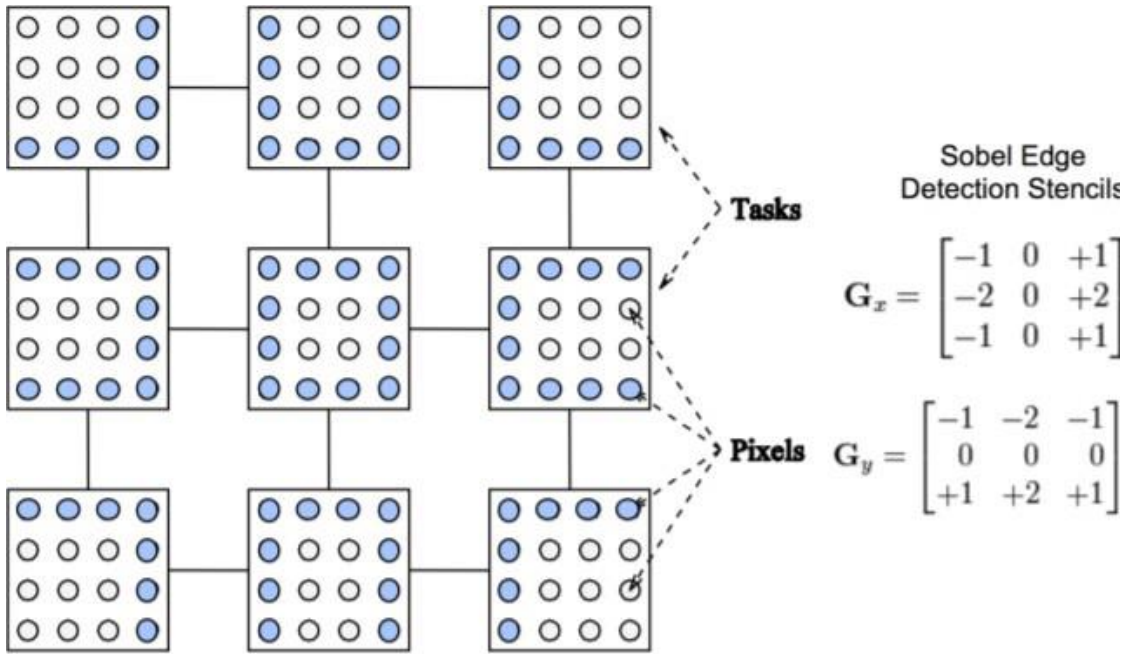
– What: shared data or synchronizations, and sizes of the media
– When: the timing
– Who: with which task(s), and overall topology/patterns
– Do we know details of the above three before execution
– How: involve one or both?

• Regular interactions
  – Definite pattern of the interactions
    • E.g. a mesh or ring
  – Can be exploited for efficient implementation.

• Irregular interactions
  – lack well-defined topologies
  – Modeled as a graph

Example of Regular Static Interaction:

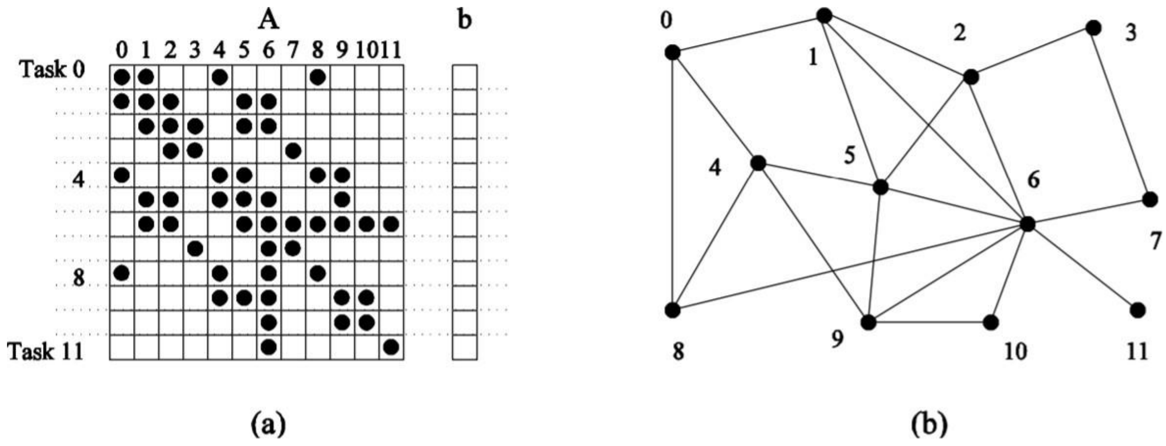
Image processing algorithms: dithering, edge detection
• Nearest neighbor interactions on a 2D mesh



$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

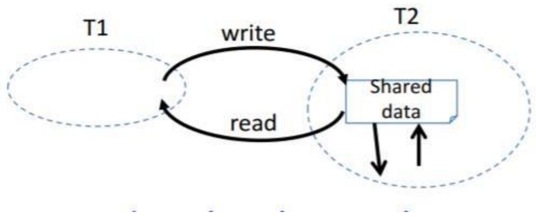
Example of Irregular Static Interaction
Sparse matrix vector multiplication



Characteristics of Task Interactions:

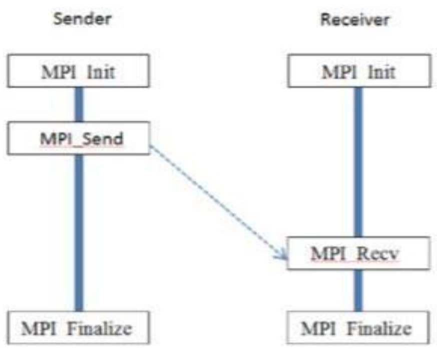
Aspects of interactions

– What: shared data or synchronizations, and sizes of the media



• Read-only interactions
  – Tasks only read data items associated with other tasks
• Read-write interactions
  – Read, as well as modify data items associated with other tasks.
  – Harder to code
• Require additional synchronization primitives
  – to avoid read-write and write-write ordering races

Aspects of interactions
  – What: shared data or synchronizations, and sizes of the media
  – When: the timing
  – Who: with which task(s), and overall topology/patterns
  – Do we know details of the above three before execution
  – How: involve one or both?
    • The implementation concern, implicit or explicit
• One-sided
  – initiated & completed independently by 1 of 2 interacting tasks
    • GET and PUT
• Two-sided
  – both tasks coordinate in an interaction
    • SEND + RECV



Mapping Techniques for Load Balancing – Static and Dynamic Mapping:

• Parallel algorithm design

– Program decomposed

– Characteristics of task and interactions identified

Assign large amount of concurrent tasks to equal or relatively small amount of processes for execution

• Though often we do 1:1 mapping

Goal of mapping:

minimize overheads

– There is cost to do parallelism

• Interactions and idling(serialization)

• Contradicting objectives: interactions vs idling

– Idling (serialization) ñ: insufficient parallelism

– Interactions ñ: excessive concurrency

– E.g. Assigning all work to one processor trivially minimizes interaction at the expense of significant idling.

Mapping Techniques for Minimum Idling:

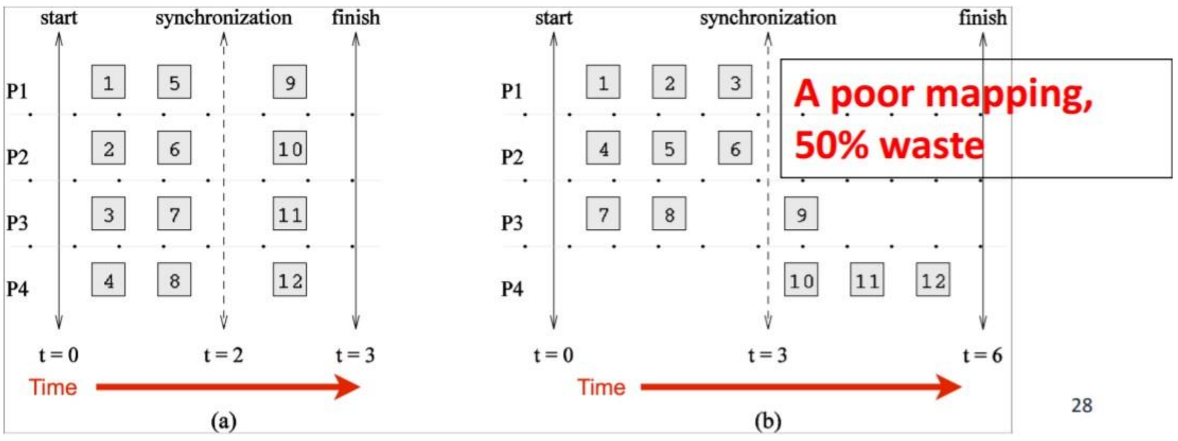
Execution: alternating stages of computation and interaction

• Mapping must simultaneously minimize idling and load balance

– Idling means not doing useful work

– Load balance: doing the same amount of work

• Merely balancing load does not minimize idling



Static or dynamic mapping:

Static Mapping

– Tasks are mapped to processes a-prior

– Need a good estimate of task sizes

– Optimal mapping may be NP complete

• Dynamic Mapping

– Tasks are mapped to processes at runtime

Because:
• Tasks are generated at runtime
• Their sizes are not known.
• Other factors determining the choice of mapping techniques
– the size of data associated with a task
– the characteristics of inter-task interactions
– even the programming models and target architectures

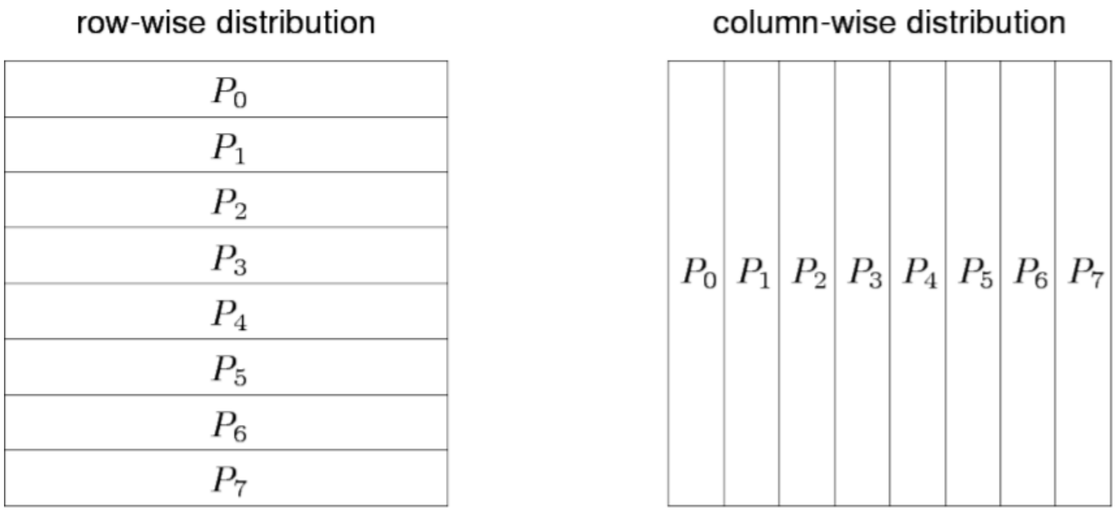
Schemes for Static Mapping:

Mappings based on data decomposition
– Mostly 1-1 mapping
• Mappings based on task graph partitioning
• Hybrid mappings

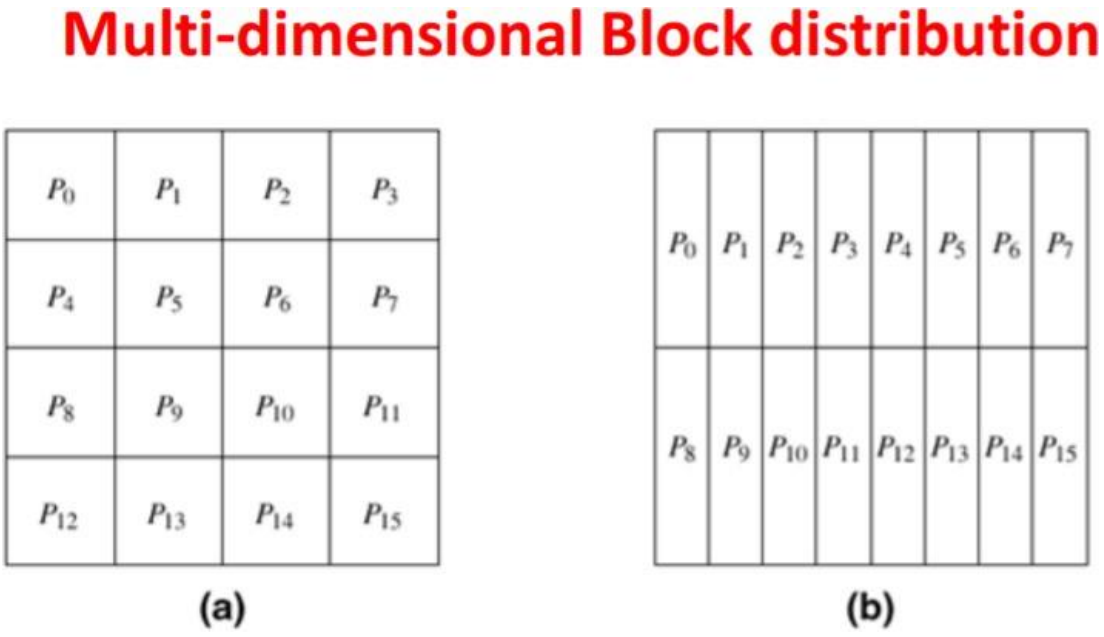
Mappings Based on Data Partitioning

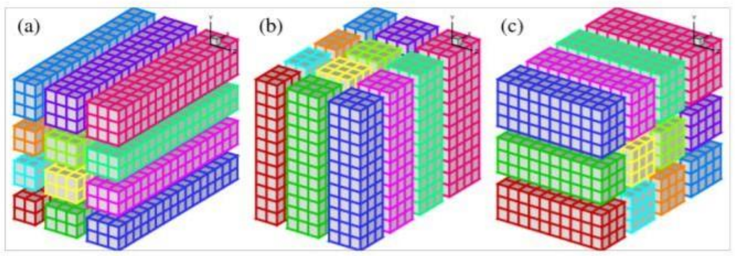
• Partition the computation using a combination of
– Data decomposition
– The ``owner-computes'' rule

Example: 1-D block distribution of 2-D dense matrix 1-1 mapping of task/data and process



Block Array Distribution Schemes:





In general, higher dimension decomposition allows the use of larger # of processes.

Block Array Distribution Schemes: Examples

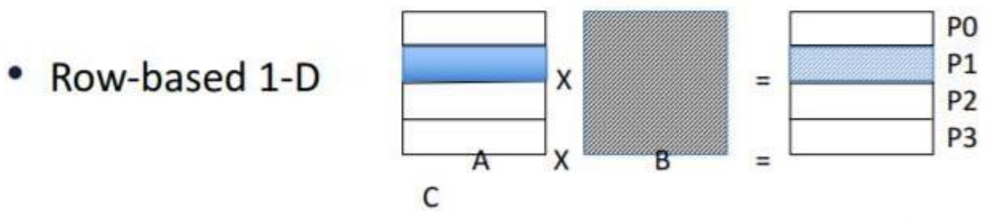
Block Array Distribution Schemes: Examples
Multiplying two dense matrices: A * B = C

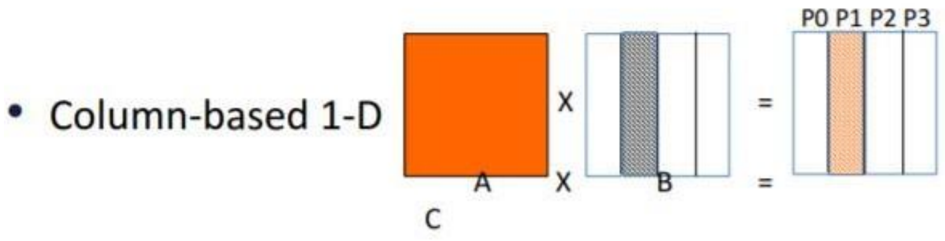
- Partition the output matrix C using a block decomposition
  - Load balance: Each task compute the same number of elements of C
- Note: each element of C corresponds to a single dot product
  - The choice of precise decomposition: 1-D (row/col) or 2-D
- Determined by the associated communication overhead

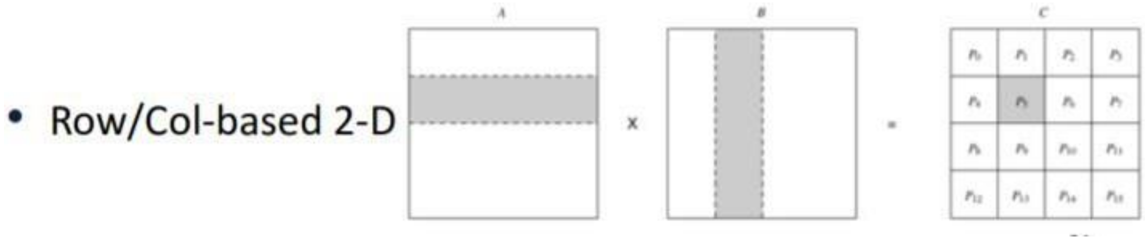
$$\begin{bmatrix} A(11) & A(12) & A(13) \\ A(21) & A(22) & A(23) \\ A(31) & A(32) & A(33) \end{bmatrix} * \begin{bmatrix} B(11) & B(12) & B(13) \\ B(21) & B(22) & B(23) \\ B(31) & B(32) & B(33) \end{bmatrix} = \begin{bmatrix} C(11) & C(12) & C(13) \\ C(21) & C(22) & C(23) \\ C(31) & C(32) & C(33) \end{bmatrix}$$

```
C(11) = A(11)*B(11) + A(12)*B(21) + A(13)*B(31)
C(21) = A(21)*B(11) + A(22)*B(21) + A(23)*B(31)
C(31) = A(31)*B(11) + A(32)*B(21) + A(33)*B(31)

C(12) = A(11)*B(12) + A(12)*B(22) + A(13)*B(32)
C(22) = A(21)*B(12) + A(22)*B(22) + A(23)*B(32)
C(32) = A(31)*B(12) + A(32)*B(22) + A(33)*B(32)

C(13) = A(11)*B(13) + A(12)*B(23) + A(13)*B(33)
C(23) = A(21)*B(13) + A(22)*B(23) + A(23)*B(33)
C(33) = A(31)*B(13) + A(32)*B(23) + A(33)*B(33)
```

Block Distribution and Data Sharing for Dense Matrix Multiplication:





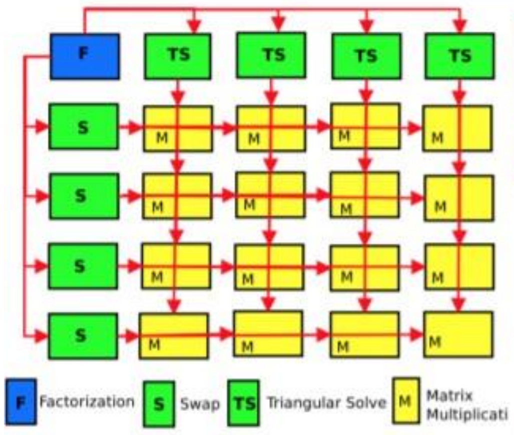


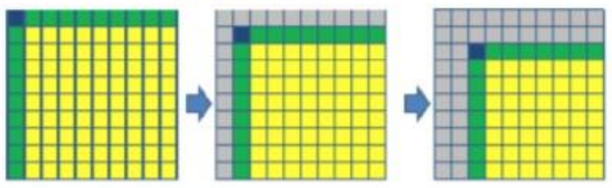
Cyclic and Block Cyclic Distributions

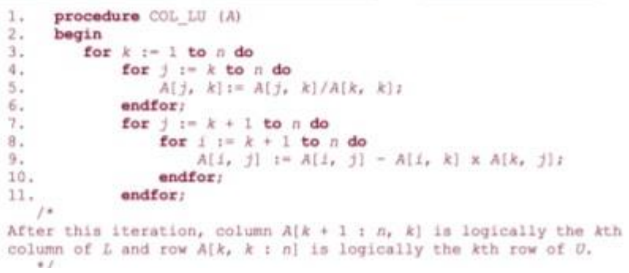
• Consider a block distribution for LU decomposition (Gaussian Elimination)

  – The amount of computation per data item varies

  – Block decomposition would lead to significant load imbalance







```
procedure COL_LU (A)
begin
   for k := 1 to n do
       for j := k to n do
           A[j, k]:= A[j, k]/A[k, k];
       endfor;
       for j := k + 1 to n do
           for i := k + 1 to n do
               A[i, j] := A[i, j] - A[i, k] x A[k, j];
           endfor;
       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.
    */
```

LU Factorization of a Dense Matrix:

A decomposition of LU factorization into 14 tasks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

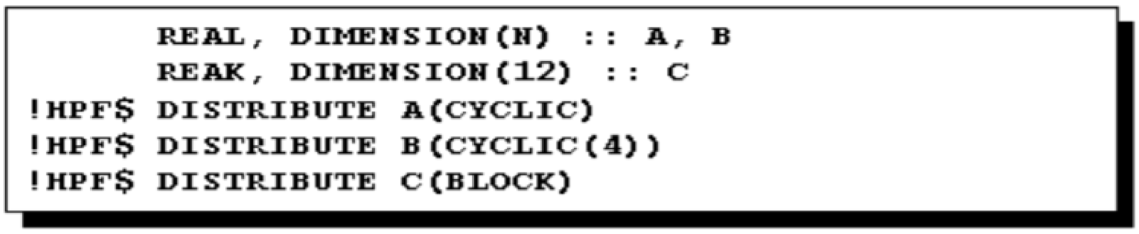
Block Distribution for LU:

Notice the significant load imbalance

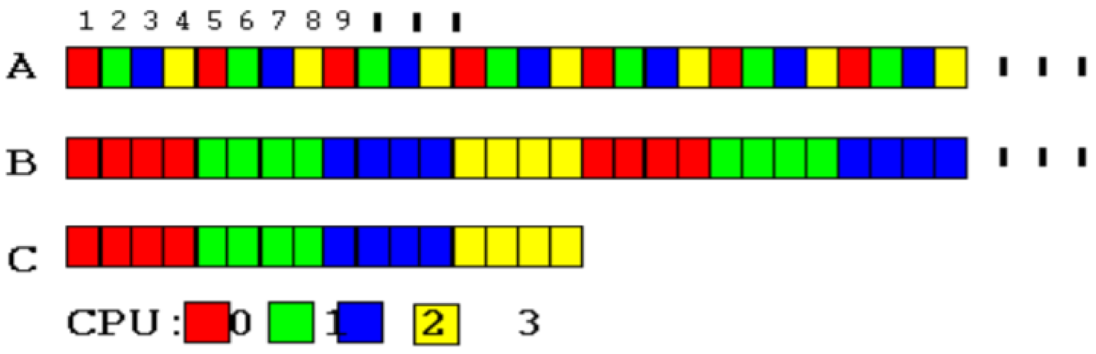
P_0 T_1	P_3 T_4	P_6 T_5
P_1 T_2	P_4 T_6 T_{10}	P_7 T_8 T_{12}
P_2 T_3	P_5 T_7 T_{11}	P_8 T_9 T_{13} T_{14}

Block Cyclic Distributions:

- Variation of the block distribution scheme
  - – Partition an array into many more blocks (i.e. tasks) than the number of available processes.
  - – Blocks are assigned to processes in a round-robin manner so that each process gets several non-adjacent blocks.
  - – N-1 mapping of tasks to processes
- Used to alleviate the load-imbalance and idling problems.



```
      REAL, DIMENSION(N) :: A, B
      REAK, DIMENSION(12) :: C
!HPF$ DISTRIBUTE A(CYCLIC)
!HPF$ DISTRIBUTE B(CYCLIC(4))
!HPF$ DISTRIBUTE C(BLOCK)
```

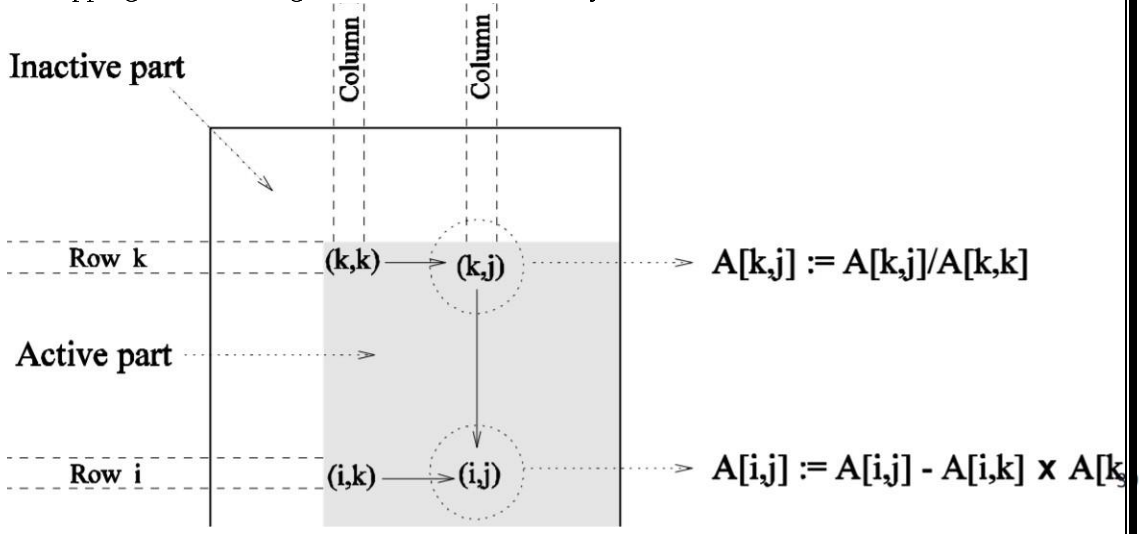


Block-Cyclic Distribution for Gaussian Elimination:

- Active submatrix shrinks as elimination progresses
- Assigning blocks in a block-cyclic fashion

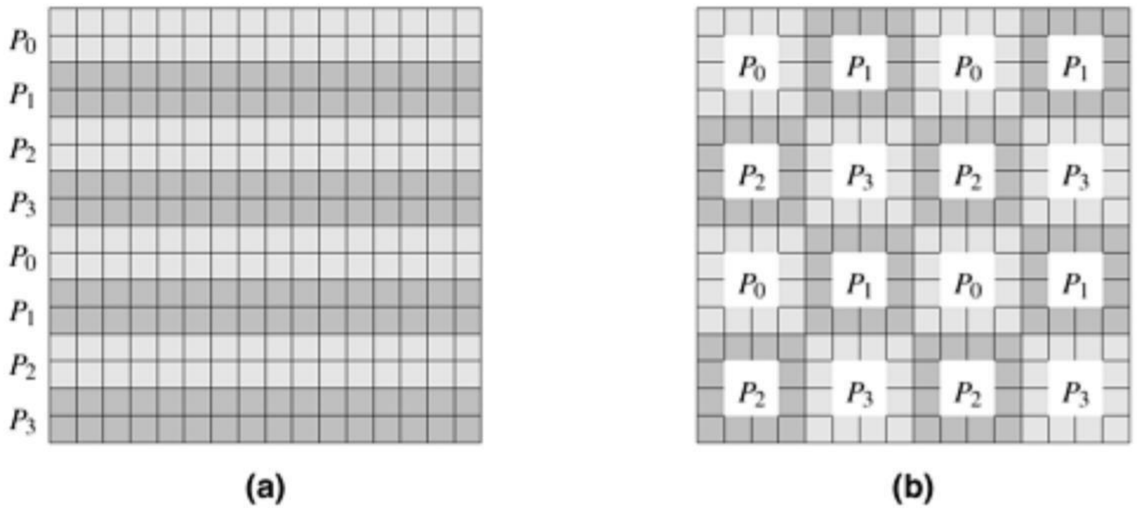
  – Each PEs receives blocks from different parts of the matrix

  – In one batch of mapping, the PE doing the most will most likely receive the least in the next batch.



Block-Cyclic Distribution:

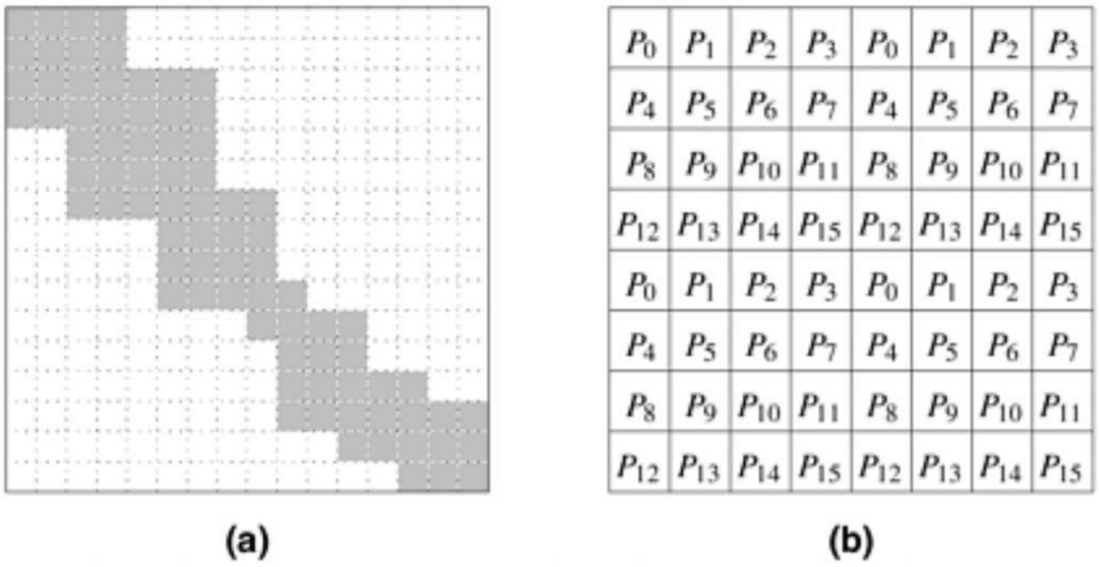
- ➢ A cyclic distribution: a special case with block size = 1
  - • A block distribution: a special case with block size = n/p
  - • n is the dimension of the matrix and p is the #of processes.



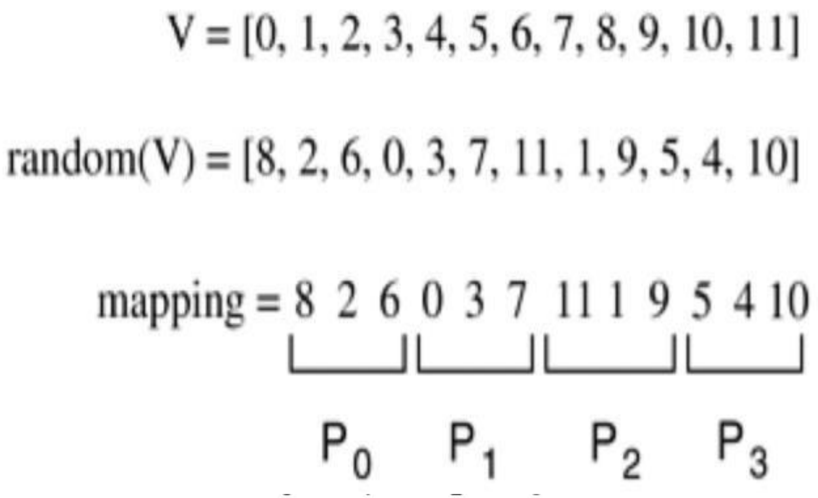
Block Partitioning and Random Mapping:

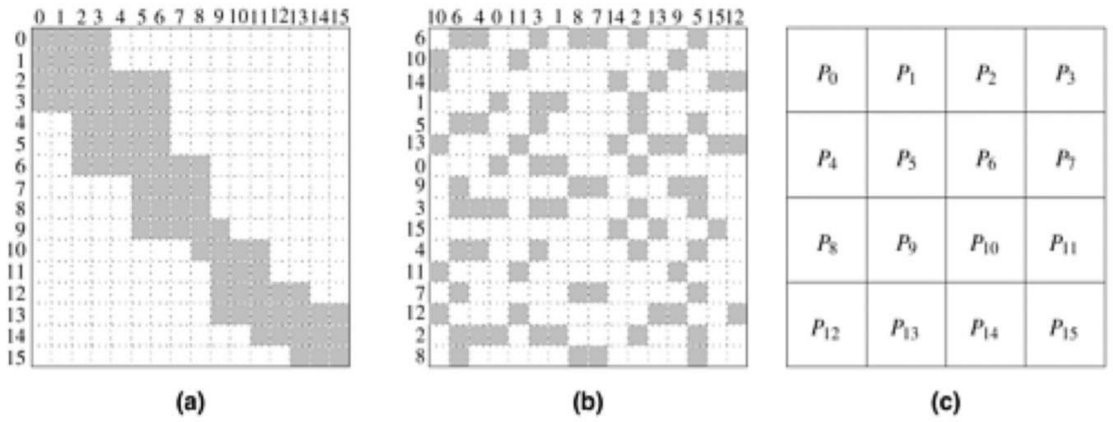
Sparse matrix computations

• Load imbalance using block-cyclic partitioning/mapping
  – more non-zero blocks to diagonal processes P0, P5, P10, and P15 than others
  – P12 gets nothing



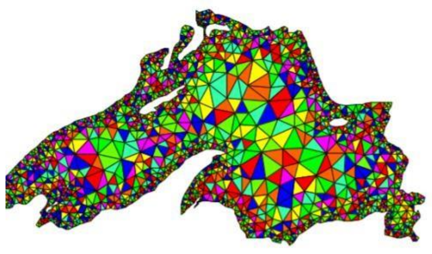
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}


V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
random(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]
mapping = 8 2 6 0 3 7 11 1 9 5 4 10
P0
P1
P2
P3

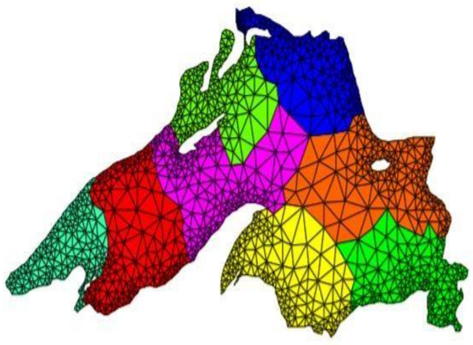


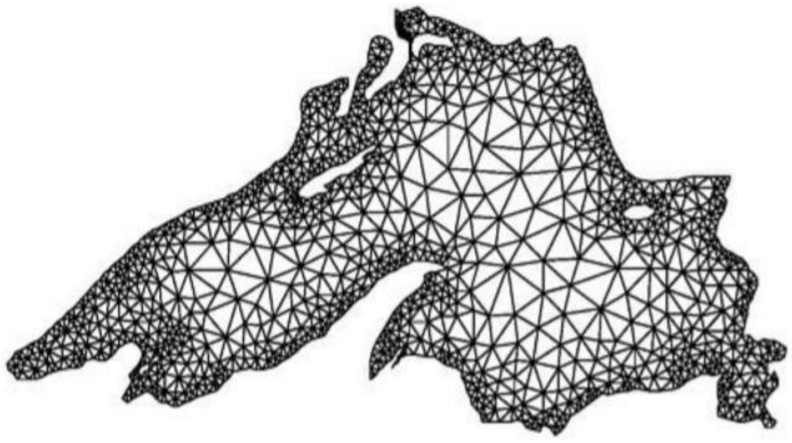
Graph Partitioning Based Data Decomposition:

- Array-based partitioning and static mapping
  - – Regular domain, i.e. rectangular, mostly dense matrix
  - – Structured and regular interaction patterns
  - – Quite effective in balancing the computations and minimizing the interactions
- Irregular domain
  - – Spars matrix-related
  - – Numerical simulations of physical phenomena
- Car, water/blood flow, geographic
- Partition the irregular domain so as to
  - – Assign equal number of nodes to each process
  - – Minimizing edge count of the partition.



Random Partitioning


Partitioning for minimum edge-cut.



Mappings Based on Task Partitioning:

- Schemes for Static Mapping
  - – Mappings based on data partitioning
    - • Mostly 1-1 mapping
  - – Mappings based on task graph partitioning
  - – Hybrid mappings
    - • Data partitioning
      - – Data decomposition and then 1-1 mapping of tasks to PEs

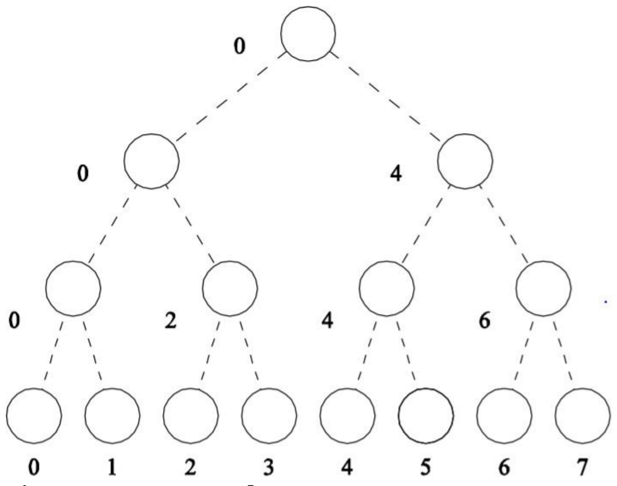
Partitioning a given task-dependency graph across processes

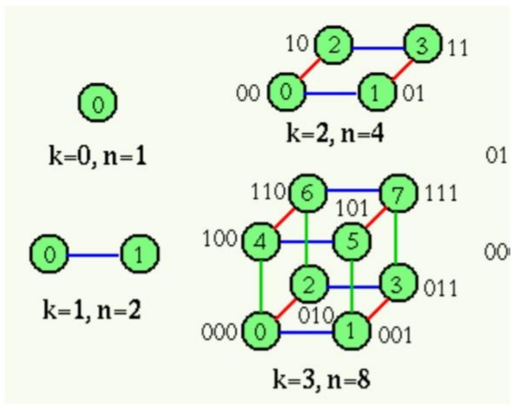
- • An optimal mapping for a general task-dependency graph
  - – NP-complete problem.
- • Excellent heuristics exist for structured graphs.

Mapping a Binary Tree Dependency Graph:

Mapping dependency graph of quick sort to processes in a hypercube

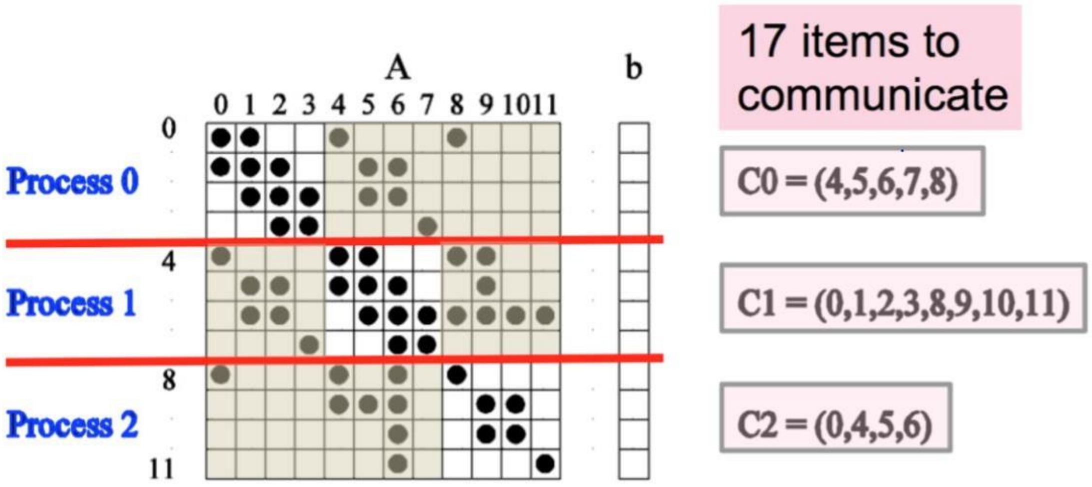
- • Hypercube: n-dimensional analogue of a square and a cube
  - – node numbers that differ in 1 bit are adjacent



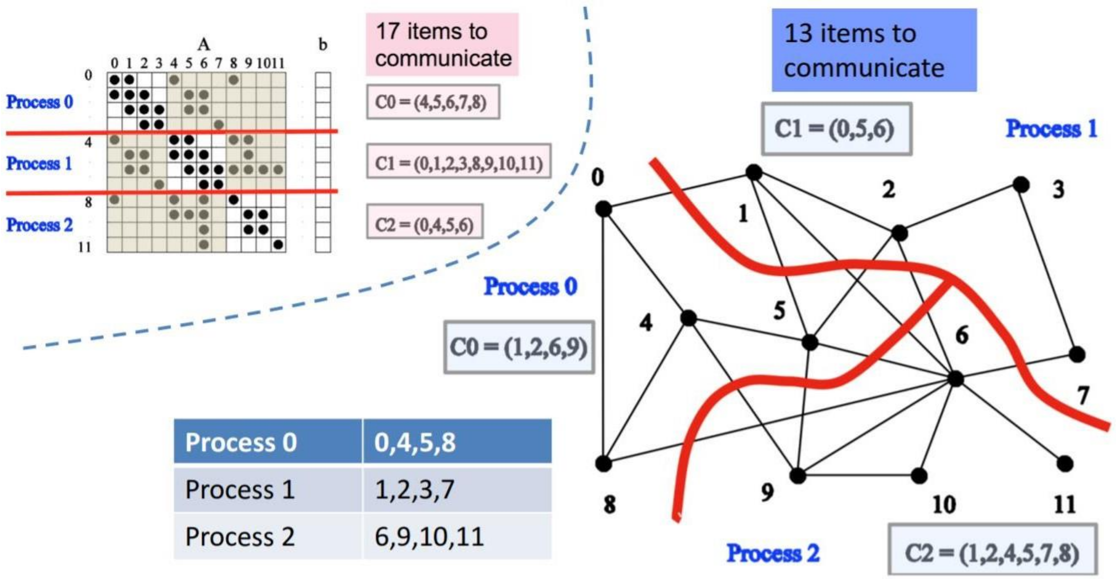


Mapping a Sparse Graph

Sparse matrix vector multiplication Using data partitioning



Sparse matrix vector multiplication using task graph partitioning



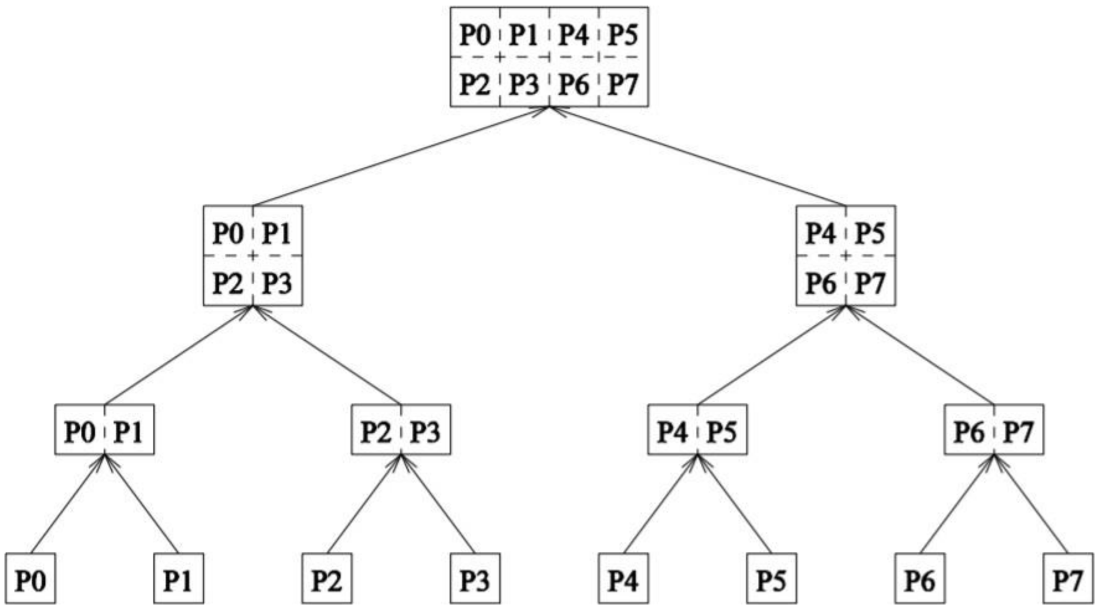
Process 0	0,4,5,8
Process 1	1,2,3,7
Process 2	6,9,10,11

Hierarchical/Hybrid Mappings:

• A single mapping is inadequate.

– E.g. task graph mapping of the binary tree (quicksort) cannot use a large number of processors.

•Hierarchical mapping – Task graph mapping at the top level – Data partitioning within each level.



Schemes for Dynamic Mapping:

Also referred to as dynamic load balancing

– Load balancing is the primary motivation for dynamic mapping.

• Dynamic mapping schemes can be

– Centralized

– Distributed

Centralized Dynamic Mapping:

Processes are designated as masters or slaves

– Workers (slave is politically incorrect)

• General strategies

– Master has pool of tasks and as central dispatcher

– When one runs out of work, it requests from master for more work.

• Challenge

– When process # increases, master may become the bottleneck.

• Approach

– Chunk scheduling: a process picks up multiple tasks at once

– Chunk size:

• Large chunk sizes may lead to significant load imbalances as well

• Schemes to gradually decrease chunk size as the computation progresses.

Distributed Dynamic Mapping:

All processes are created equal

– Each can send or receive work from others

• Alleviates the bottleneck in centralized schemes.

• Four critical design questions:

– how are sending and receiving processes paired together

– who initiates work transfer

– how much work is transferred

– when is a transfer triggered?

• Answers are generally application specific.

• Work stealing

CUDA programming model: Overview of CUDA, Isolating data to be used by parallelized code, API function to allocate memory on parallel computing device, to transfer data.

What is CUDA programming model?

The CUDA programming model provides **an abstraction of GPU architecture** that acts as a bridge between an application and its possible implementation on GPU hardware. ... The GPU is called a device and GPU memory likewise called device memory.

UNIT-III

Analytical Modeling of Parallel Programs: Sources of Overhead in Parallel Programs, Performance Metrics for Parallel Systems, The Effect of Granularity on Performance, Scalability of Parallel Systems, Minimum Execution Time and Minimum Cost Optimal Execution Time

Analytical Modeling - Basics

• A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).

• The asymptotic runtime of a sequential program is identical on any serial platform.

• The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.

• An algorithm must therefore be analyzed in the context of the underlying platform.

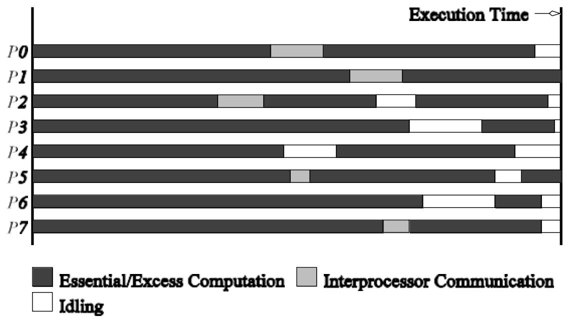
• A parallel system is a combination of a parallel algorithm and an underlying platform.

- A number of performance measures are intuitive.
- Wall clock time - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. But how does this scale when the number of processors is changed of the program is ported to another machine altogether?
- How much faster is the parallel version? This begs the obvious followup question - whats the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look
- Raw FLOP count - What good are FLOP counts when they dont solve a problem?

Sources of Overhead in Parallel Programs

- If I use two processors, shouldnt my program run twice as fast?

No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.



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

•Interprocess interactions: Processors working on any non-trivial parallel problem will need to talk to each other.

•Idling: Processes may idle because of load imbalance, synchronization, or serial components.

- Excess Computation: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Performance Metrics for Parallel Systems: Execution Time

- 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 the first processor starts to the moment the last processor finishes execution.

  We denote the serial runtime by and the parallel runtime by T_P.

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_S is the serial time.
- Observe that T_{all} - T_S is then the total time spend by all processors combined in non-useful work. This is called the total overhead.
- The total time collectively spent by all the processing elements

  T_{all} = p T_P (p is the number of processors).
- The overhead function (T_o) is therefore given by

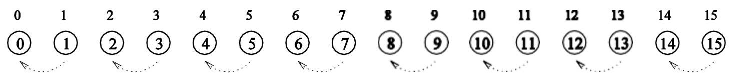
$$T_o = p\,T_P - T_S$$

Performance Metrics for Parallel Systems: Speedup

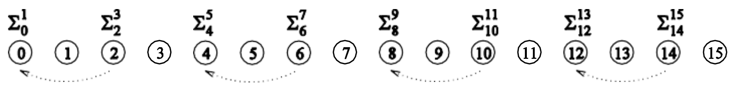
- What is the benefit from parallelism?
- Speedup (*S*) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with *p* identical processing elements.

Performance Metrics: Example

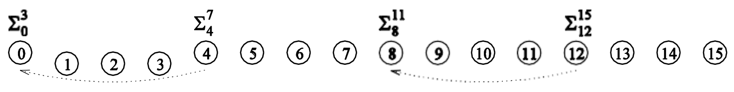
- Consider the problem of adding *n* numbers by using *n* processing elements.
- If *n* is a power of two, we can perform this operation in log *n* steps by propagating partial sums up a logical binary tree of processors.



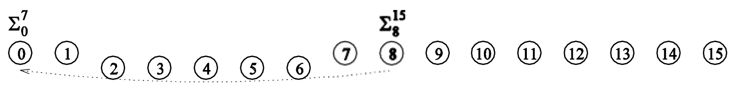
(a) Initial data distribution and the first communication step


Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}

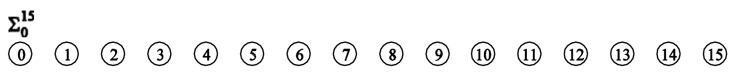
(b) Second communication step


Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}

(c) Third communication step



(d) Fourth communication step



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

Computing the globalsum of 16 partial sums using 16 processing elements . Σ^{j}_{i} denotes the sum of numbers with consecutive labels from ***i*** to ***j***.

- If an addition takes constant time, say, t_c and communication

  of a single word takes time $t_s + t_w$, we have the parallel time
  $T_P = \Theta(\log n)$
- We know that $T_S = \Theta(n)$

Speedup ***S*** is given by $S = \Theta(n / \log n)$

Performance Metrics: Speedup

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.
- For the purpose of computing speedup, we always consider the best sequential program as the baseline.

Performance Metrics: Speedup Example

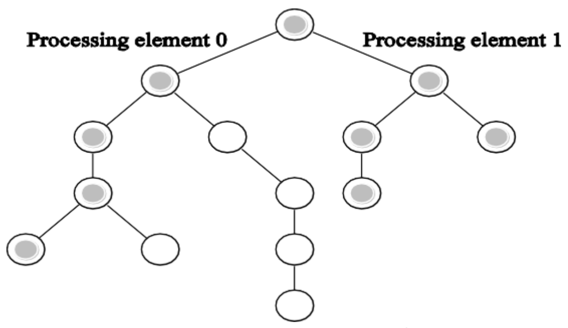
- Consider the problem of parallel bubble sort.
- The serial time for bubblesort is 150 seconds.
- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds.
- The speedup would appear to be 150/40 = 3.75.
- But is this really a fair assessment of the system?
- What if serial quicksort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system.

Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates).
- Speedup, in theory, should be upper bounded by p - after all, we can only expect a p-fold speedup if we use times as many resources.
- A speedup greater than p is possible only if each processing element spends less than time T_S / p solving the problem.
- In this case, a single processor could be timeslided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Super linear Speedups

One reason for super linearity is that the parallel version does less work than corresponding serial algorithm**.**



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

Performance Metrics: Super linear Speedups

Resource-based super linearity: The higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore super linearity.

Example: A processor with 64KB of cache yields an 80% hit ratio. If two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory.

If DRAM access time is 100 ns, cache access time is 2 ns, and remote memory access time is 400ns, this corresponds to a speedup of 2.43!

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed
- Mathematically, it is given by

$$E=S/P \text{---------} \rightarrow (2)$$

- Following the bounds on speedup, efficiency can be as low as 0 and as high as 1.

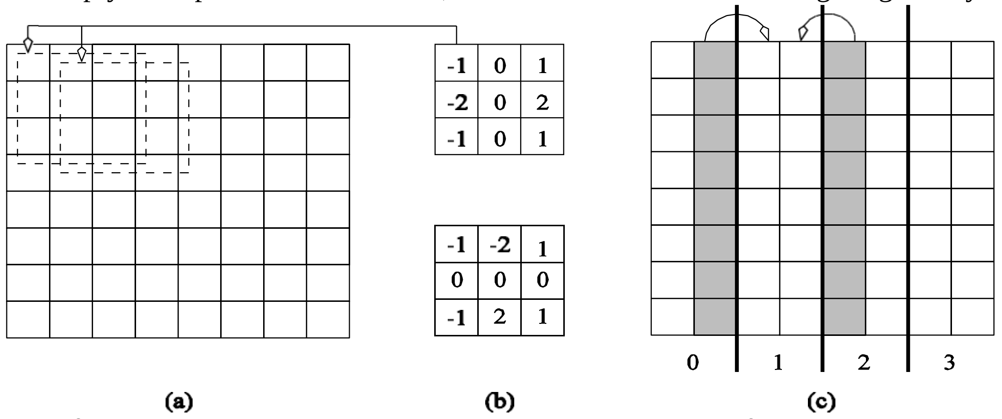
Performance Metrics: Efficiency Example

- The speedup of adding numbers on processors is given by
- Efficiency is given by

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n} = \Theta\left(\frac{1}{\log n}\right)$$

Parallel Time, Speedup, and Efficiency Example

Consider the problem of edge-detection in images. The problem requires us to apply a *3* x *3* template to each pixel. If each multiply-add operation takes time t_c, the serial time for an n x n image is given by $T_S = t_c n^2$.



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

- One possible parallelization partitions the image equally into vertical segments, each with $\boldsymbol{n^2 / p}$ pixels.
- The boundary of each segment is ***2n*** pixels. This is also the number of pixel values that will have to be

communicated. This takes time $2(t_s + t_w n)$.

- Templates may now be applied to all n^2 / p pixels in time

$T_S = 9\, t_c n^2 / p$.

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

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

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

- The total time for the algorithm is therefore given by:

Cost of a Parallel System:

- Cost is the product of parallel runtime and the number of processing elements used ($p \times T_P$).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $E = T_S / p\, T_P$, for cost optimal systems, $E = O(1)$.
- Cost is sometimes referred to as *work* or *processor-time product*.

Consider the problem of adding numbers on processors.

- We have, $T_P = \log n$ (for $p = n$).
- The cost of this system is given by $p\, T_P = n \log n$.
- Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.

Impact of Non-Cost Optimality:

Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$.

- Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n / \log n$ and $1 / \log n$, respectively.
- The $p\, T_P$ product of this algorithm is $n (\log n)^2$.
- This algorithm is not cost optimal but only by a factor of *log n*.
- If $p < n$, assigning n tasks to p processors gives $T_P = n (\log n)^2 / p$.
- The corresponding speedup of this formulation is $p / \log n$.
- This speedup goes down as the problem size n is increased for a given p !

Effect of Granularity on Performance:

- Often, using fewer processors improves performance of parallel systems.
- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm

is called *scaling down* a parallel system.

- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.
- Since 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.
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might talk to each other. This is the basic reason for the improvement from building granularity.

Building Granularity: Example

- Consider the problem of adding n numbers on p processing elements such that $p < n$ and both n and p are powers of 2.
- Use the parallel algorithm for n processors, except, in this case, we think of them as virtual processors.
- Each of the p processors is now assigned n / p virtual processors.
- 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.

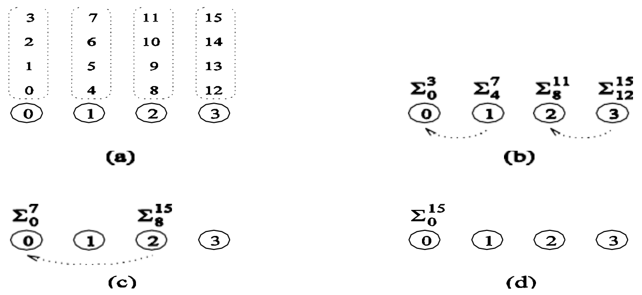
Subsequent log n - log p steps do not require any communication.

- The overall parallel execution time of this parallel system is

  $\Theta\ (\ (\boldsymbol{n} / \boldsymbol{p})\ \mathbf{log}\ \boldsymbol{p})$.
- The cost is Θ ($\boldsymbol{n}$ **log** $\boldsymbol{p}$), which is asymptotically higher than the Θ ($\boldsymbol{n}$) cost of adding $\boldsymbol{n}$ numbers sequentially. Therefore, the parallel system is not cost-optimal.

Can we build granularity in the example in a cost-optimal fashion?

- Each processing element locally adds its $\boldsymbol{n} / \boldsymbol{p}$ numbers in time Θ ($\boldsymbol{n} / \boldsymbol{p}$).
- The $\boldsymbol{p}$ partial sums on $\boldsymbol{p}$ processing elements can be added in time $\Theta(\boldsymbol{n} / \boldsymbol{p})$.



3 2 1 0
7 6 5 4
11 10 9 8
15 14 13 12
0 1 2 3
(a)
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(b)
Σ_0^7 Σ_8^{15}
(c)
Σ_0^{15}
(d)

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

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

- The parallel runtime of this algorithm is
- The cost is

$$n = \Omega(p \log p)$$

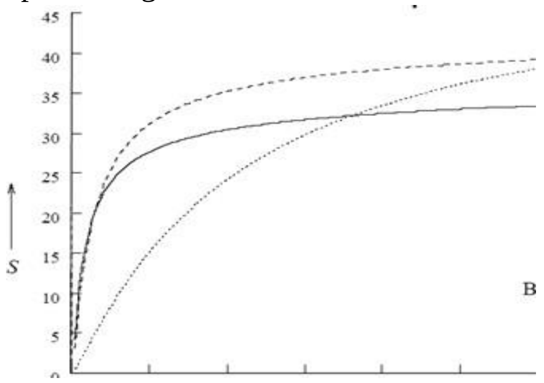
This is cost-optimal, so long as

$$\Theta(n + p \log p) \quad !$$

Scalability of Parallel Systems:

How do we extrapolate performance from small problems and small systems to larger problems on larger configurations?

Consider three parallel algorithms for computing an *n*-point Fast Fourier Transform (FFT) on 64 processing elements.



A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$.

Clearly, it is difficult to infer scaling characteristics from observations on small datasets on small machines.

- For a given problem size (i.e., the value of T_S remains constant), as we increase the number of processing elements, T_o increases.
- The overall efficiency of the parallel program goes down. This is the case for all parallel programs.
- Consider the problem of adding numbers on processing elements.

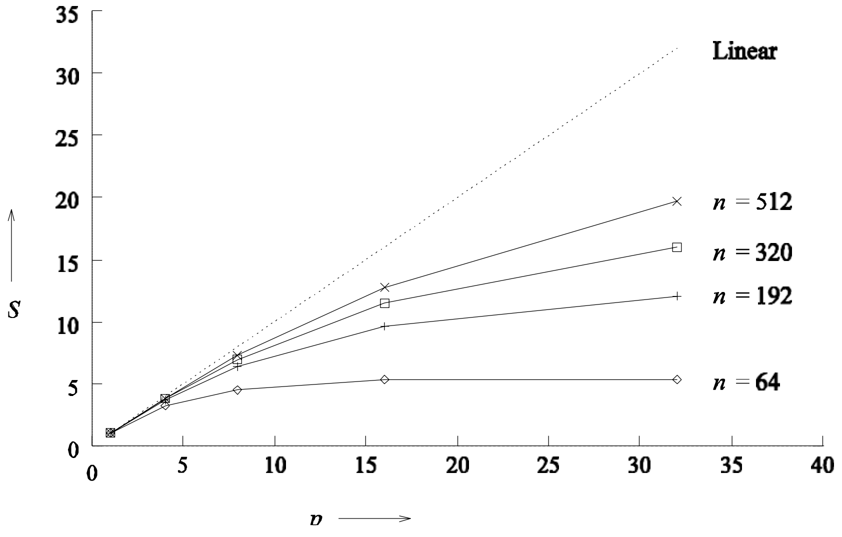
We have seen that:

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

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

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

Plotting the speedup for various input sizes gives us:



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

Speedup tends to saturate and efficiency drops as a consequence of Amdahl's law

Scaling Characteristics of Parallel Programs:

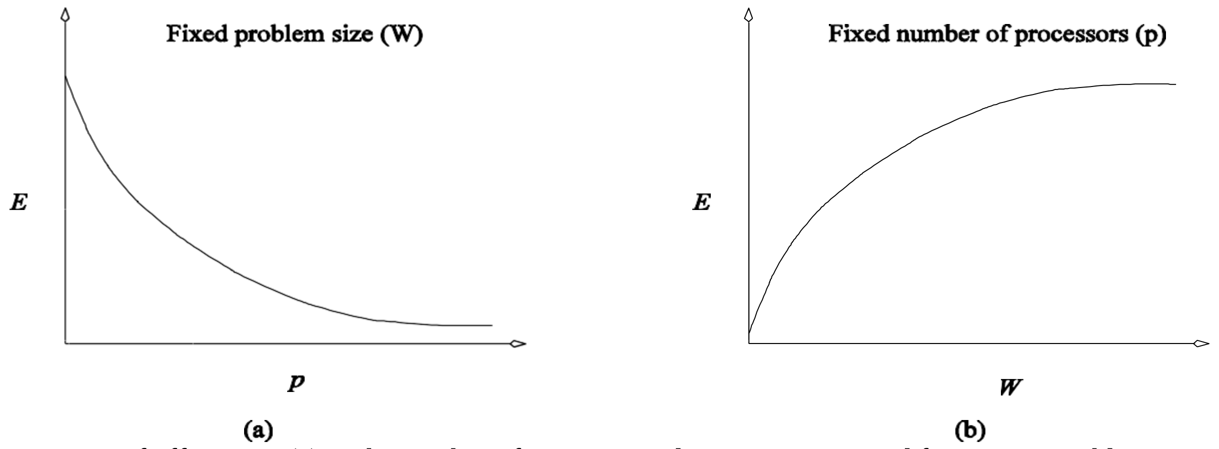
- Total overhead function T_o is a function of both problem size T_s and the number of processing elements p.
- In many cases, T_o grows sublinearly with respect to T_s.
- In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.
- For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant.

We call such systems *scalable* parallel systems.

- Recall that cost-optimal parallel systems have an efficiency of Θ(1).
- Scalability and cost-optimality are therefore related.
- A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.

Isoefficiency Metric of Scalability:

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.



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

- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?
- This rate determines the scalability of the system. The slower this rate, the better.
- Before we formalize this rate, we define the problem size W as the asymptotic number of operations associated with the best serial algorithm to solve the problem**.**

We can write parallel runtime as:

$$T_P = \frac{W + T_o(W,p)}{p} \qquad (8)$$

The resulting expression for speedup is

$$\begin{aligned} S &= \frac{W}{T_P} \\ &= \frac{Wp}{W + T_o(W,p)}. \qquad (9) \end{aligned}$$

Finally, we write the expression for efficiency as

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W,p)} \end{aligned}$$

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

For a desired value E of efficiency**,**

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

If $\boldsymbol{K} = \boldsymbol{E} / (1 - \boldsymbol{E})$ is a constant depending on the efficiency to be maintained, since $\boldsymbol{T_o}$ is a function of $\boldsymbol{W}$ and $\boldsymbol{p}$, we have

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

- The problem size ***W*** can usually be obtained as a function of ***p*** by algebraic manipulations to keep efficiency constant.
- This function is called the *isoefficiency function.*
- This 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

Isoefficiency Metric: Example

- The overhead function for the problem of adding n numbers on p processing elements is approximately $2p \log p$.

Substituting T_o by $2p \log p$, we get

$$W \qquad K2p\log p.$$

Thus, the asymptotic isoefficiency function for this parallel system is

.

If the number of processing elements is increased from ***p*** to ***p'***, the problem size (in this case, n) must be increased by a factor of (***p'* log *p'***) / (***p* log *p***) to get the same efficiency as on ***p*** processing elements.

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

Consider a more complex example where

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

- Using only the first term of ***T***T_o in Equation 12, we get

$$W = Kp^{3/2}. \qquad (14)$$

- Using only the second term, Equation 12 yields the following relation between ***W*** and ***p***:

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

The larger of these two asymptotic rates determines the isoefficiency. This is given by $\Theta(p^3)$

Cost-Optimality and the Isoefficiency Function

- A parallel system is cost-optimal if and only if

$$pT_P = \Theta(W). \qquad (16)$$

- From this, we have:

$$\begin{aligned} W + T_o(W,p) &= \Theta(W) \\ T_o(W,p) &= O(W) \qquad (17) \\ W &= \Omega(T_o(W,p)) \qquad (18) \end{aligned}$$

If we have an isoefficiency function ***f***(***p***), then it follows that the relation ***W*** = Ω(***f***(***p***)) must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

- For a problem consisting of ***W*** units of work, no more than ***W*** processing elements can be used cost-optimally.

- The problem size must increase at least as fast as Θ(***p***) to maintain fixed efficiency; hence, Ω(***p***) is the asymptotic lower bound on the isoefficiency function.

Degree of Concurrency and the Isoefficiency Function

- The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*.

- If ***C***(***W***) is the degree of concurrency of a parallel algorithm, then for a problem of size ***W***, no more than ***C***(***W***) processing elements can be employed effectively.

Degree of Concurrency and the Isoefficiency Function: Example

Consider solving a system of equations in   variables by using Gaussian elimination ($W = \Theta(n^3)$)

- The n variables must be eliminated one after the other, and eliminating each variable requires $\Theta(n^2)$ computations.
- At most $\Theta(n^2)$ processing elements can be kept busy at any time.
- Since $W = \Theta(n^3)$ for this problem, the degree of concurrency $C(W)$ is $\Theta(W^{2/3})$.

Given p processing elements, the problem size should be at least $\Omega(p^{3/2})$ to use them all.

Minimum Execution Time and Minimum Cost-Optimal Execution Time:

Often, we are interested in the minimum time to solution.

- We can determine the minimum parallel runtime T_P^{min} for a given W by differentiating the expression for T_P w.r.t. p and equating it to zero.

$$\frac{d}{dp} T_P = 0 \qquad (19)$$

- If p_0 is the value of p as determined by this equation, $T_P(p_0)$ is the minimum parallel time.

  Minimum Execution Time: Example

  Consider the minimum execution time for adding n numbers.

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

= (20)

Setting the derivative w.r.t. p to zero, we have $p = n/2$. The corresponding runtime is

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

= (21)

(One may verify that this is indeed a min by verifying that the second derivative is positive).
Note that at this point, the formulation is not cost-optimal.

Minimum Cost-Optimal Parallel Time:

- Let $T_P^{cost\_opt}$ be the minimum cost-optimal parallel time.
- 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, for cost optimality, $p = O(f^{-1}(W))$.
- For cost-optimal systems, $T_P = \Theta(W/p)$, therefore**,**

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right). \qquad (22)$$

Minimum Cost-Optimal Parallel Time: Example:

Consider the problem of adding n numbers.

- The isoefficiency function $f(p)$ of this parallel system is $\Theta(p \log p)$.
- From this, we have $p \approx n / \log n$.

At this processor count, the parallel runtime is:

$$T_P^{cost\_opt} = \log n + \log\left(\frac{n}{\log n}\right)$$
$$= 2\log n - \log\log n. \quad (23)$$

Asymptotic Analysis of Parallel Programs:
Consider the problem of sorting a list of *n* numbers. The fastest serial programs for this problem run in time Θ(*n* log *n*). Consider four parallel algorithms, A1, A2, A3, and A4 as follows:

Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the pT_P product**.**

Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n}\log n$
S	$n\log n$	$\log n$	$\sqrt{n}\log n$	$\sqrt{n}$
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n\log n$	$n^{1.5}$	$n\log n$

- If the metric is speed, algorithm A1 is the best, followed by A3, A4, and A2 (in order of increasing T_P).
- In terms of efficiency, A2 and A4 are the best, followed by A3 and A1.
- In terms of cost, algorithms A2 and A4 are cost optimal, A1 and A3 are not.
- It is important to identify the objectives of analysis and to use appropriate metrics!

Other Scalability Metrics:
- A number of other metrics have been proposed, dictated by specific needs of applications.
- For real-time applications, the objective is to scale up a system to accomplish a task in a specified time bound.
- In memory constrained environments, metrics operate at the limit of memory and estimate performance under this problem growth rate.

Other Scalability Metrics: Scaled Speedup
- Speedup obtained when the problem size is increased linearly with the number of processing elements.
- If scaled speedup is close to linear, the system is considered scalable.
- If the isoefficiency is near linear, scaled speedup curve is close to linear as well.
- If the aggregate memory grows linearly in *p*, scaled speedup increases problem

size to fill memory.

Alternately, the size of the problem is increased subject to an upper-bound on parallel execution time.

Scaled Speedup: Example

- The serial runtime of multiplying a matrix of dimension *n x n* with a vector is $t_c n^2$.

For a given parallel algorithm,

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}} \qquad (24)$$

- Total memory requirement of this algorithm is $\Theta(n^2)$.
- Consider the case of memory-constrained scaling.
- We have m= $\Theta(\boldsymbol{n}^2) = \Theta(\boldsymbol{p})$.
- Memory constrained scaled speedup is given by

$$S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}}$$

- This is not a particularly scalable system

  Consider the case of time-constrained scaling.
- We have $T_P = O(n^2)$.
- Since this is constrained to be constant, $n^2 = O(p)$.
- Note that in this case, time-constrained speedup is identical to memory constrained speedup.
- This is not surprising, since the memory and time complexity of the operation are identical.
- The serial runtime of multiplying two matrices of dimension $\boldsymbol{n} \times \boldsymbol{n}$ is $\boldsymbol{t_c n^3}$.
- The parallel runtime of a given algorithm is:

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

- The speedup ***S*** is given by:

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}} \qquad (25)$$

Consider memory-constrained scaled speedup.

- We have memory complexity m= $\Theta(\boldsymbol{n}^2) = \Theta(\boldsymbol{p})$, or $\boldsymbol{n}^2 = \boldsymbol{c} \times \boldsymbol{p}$.
- At this growth rate, scaled speedup ***S'*** is given by:
- Note that this is scalable.

$$S' = \frac{t_c (c \times p)^{1.5}}{t_c \frac{(c \times p)^{1.5}}{p} + t_s \log p + 2t_w \frac{c \times p}{\sqrt{p}}} = O(p)$$

Consider time-constrained scaled speedup.

- We have $\boldsymbol{T_P} = O(1) = O(\boldsymbol{n}^3 / \boldsymbol{p})$, or $\boldsymbol{n}^3 = \boldsymbol{c} \times \boldsymbol{p}$.

Time-constrained speedup ***S''*** is given by:

- $$S'' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + 2t_w \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

Memory constrained scaling yields better performance.

Serial Fraction *f* :

- If the serial runtime of a computation can be divided into a totally parallel and a totally serial component, we have:

$$W = T_{ser} + T_{par}.$$

- From this, we have,

$$T_P = T_{ser} + \frac{T_{par}}{p}.$$

$$T_P = T_{ser} + \frac{W - T_{ser}}{p} \qquad (26)$$

- The serial fraction ***f*** of a parallel program is defined as:

$$f = \frac{T_{ser}}{W}.$$

Therefore, we have:

$$T_P = f \times W + \frac{W - f \times W}{p}$$

$$\frac{T_P}{W} = f + \frac{1 - f}{p}$$

- Since **$S = W / T_P$**, we have

$$\frac{1}{S} = f + \frac{1 - f}{p}.$$

From this, we have

$$f = \frac{1/S - 1/p}{1 - 1/p}.$$

- If ***f*** increases with the number of processors, this is an indicator of rising overhead, and thus an indicator of poor scalability.

Serial Fraction: Example

Consider the problem of extimating the serial component of the matrix-vector product**.**

We have:

$$f = \frac{t_s p \log p + t_w n p}{t_c n^2} \times \frac{1}{p-1}$$

$$f = \frac{\frac{t_c \frac{n^2}{p} + t_s \log p + t_w n}{t_c n^2}}{1 - 1/p}$$

$$f \approx \frac{t_s \log p + t_w n}{t_c n^2}$$

Here, the denominator is the serial runtime and the numerator is the overhead.

UNIT-IV

Dense Matrix Algorithms: Matrix-Vector Multiplication, Matrix-Matrix Multiplication, Issues in Sorting on Parallel Computers, Bubble Sort and Variants, Quick Sort Algorithm.

This section addresses the problem of multiplying a dense n x n matrix A with an n x 1 vector x to yield the n x 1 result vector y. Algorithm 8.1 shows a serial algorithm for this problem. The sequential algorithm requires n^2 multiplications and additions. Assuming that a multiplication and addition pair takes unit time, the sequential run time is Equation 8.1

$$W = n^2.$$

At least three distinct parallel formulations of matrix-vector multiplication are possible, depending on whether rowwise 1-D, columnwise 1-D, or a 2-D partitioning is used.

Algorithm 8.1 A serial algorithm for multiplying an n x n matrix A with an n x 1 vector x to yield an n x 1 product vector y.

```
procedure MAT_VECT ( A, x, y)
begin
   for i := 0 to n - 1 do
   begin
      y[i]:=0;
      for j := 0 to n - 1 do
          y[i] := y[i] + A[i, j] x x[j];
   endfor;
end MAT_VECT
```

8.1.1 Rowwise 1-D Partitioning

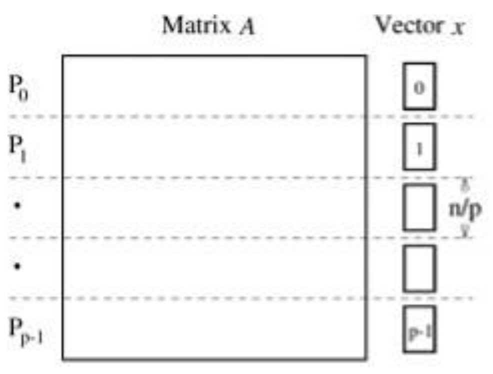
This section details the parallel algorithm for matrix-vector multiplication using rowwise block 1-D partitioning. The parallel algorithm for columnwise block 1-D partitioning is similar (Problem 8.2) and has a similar expression for parallel run time. Figure 8.1 describes the distribution and movement of data for matrix-vector multiplication with block 1-D partitioning.

Figure 8.1. Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, p = n.

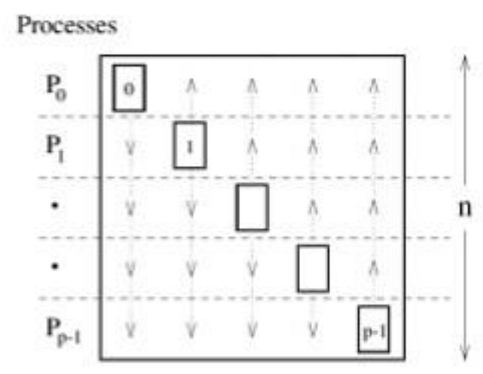
Rowwise 1-D Partitioning

This section details the parallel algorithm for matrix-vector multiplication using rowwise block 1-D partitioning. The parallel algorithm for columnwise block 1-D partitioning is similar (Problem 8.2) and has a similar expression for parallel run time. Figure 8.1 describes the distribution and movement of data for matrix-vector multiplication with block 1-D partitioning.

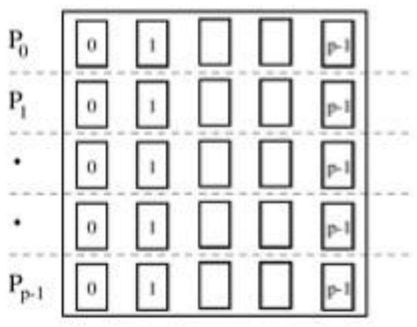
Figure 8.1. Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, p = n.



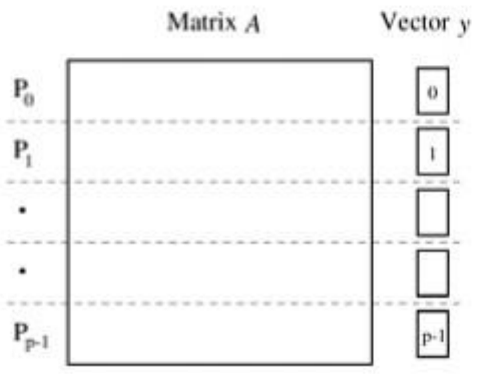
(a) Initial partitioning of the matrix and the starting vector x



(b) Distribution of the full vector among all the processes by all-to-all broadcast



(c) Entire vector distributed to each process after the broadcast



(d) Final distribution of the matrix and the result vector y

One Row Per Process First

consider the case in which the n x n matrix is partitioned among n processes so that each process stores one complete row of the matrix. The n x 1 vector x is distributed such that each process owns one of its elements. The initial distribution of the matrix and the vector for rowwise block 1-D partitioning is shown in Figure 8.1(a). Process P_i initially owns x[i] and A[i, 0], A[i, 1], ..., A[i, n-1] and is responsible for computing y[i]. Vector x is multiplied with each row of the matrix (Algorithm 8.1); hence, every process needs the entire vector. 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. Figure 8.1(b) illustrates this communication step. After the vector x is distributed among the processes (Figure 8.1(c)), process P_i computes $y[i] = \Sigma_{j=0}^{n-1}(A[i, j] \times x[j])$ (lines 6 and 7 of Algorithm 8.1). As Figure 8.1(d) shows, the result vector y is stored exactly the way the starting vector x was stored.

Parallel Run Time:

Starting with one vector element per process, the all-to-all broadcast of the vector elements among n processes requires time Q(n) on any architecture (Table 4.1). The multiplication of a single row of A with x is also performed by each process in time Q(n). Thus, the entire procedure is completed by n processes in time Q(n), resulting in a process-time product of Q(n^2). The parallel algorithm is cost-optimal because the complexity of the serial algorithm is Q(n^2).

Using Fewer than n Processes

Consider the case in which p processes are used such that p < n, and the matrix is partitioned among the processes by using block 1-D partitioning. Each process initially stores n/p complete rows of the matrix and a portion of the vector of size n/p. Since the vector x must be multiplied with each row of the matrix, every process needs the entire vector (that is, all the portions belonging to separate processes). This again requires an all-to-all broadcast as shown in Figure 8.1(b) and (c). The all-to-all broadcast takes place among p processes and involves messages of size n/p. After this communication step, each process multiplies its n/p rows with the vector x to produce n/p elements of the result vector. Figure 8.1(d) shows that the result vector y is distributed in the same format as that of the starting vector x.

Parallel Run Time According to Table 4.1, an all-to-all broadcast of messages of size n/p among p processes takes time ts log p + tw(n/ p)(p - 1). For large p, this can be approximated by ts log p + twn. After the communication, each process spends time n2/p multiplying its n/p rows with the vector. Thus, the parallel run time of this procedure is

Equation 8.2:

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

The process-time product for this parallel formulation is n2 + ts p log p + twnp. The algorithm is cost-optimal for p = O(n).

Scalability Analysis We now derive the is o efficiency function for matrix-vector multiplication along the lines of the analysis in Section 5.4.2 by considering the terms of the overhead function one at a time. Consider the parallel run time given by Equation 8.2 for the hypercube architecture. The relation To = pTP - W gives the following expression for the overhead function of matrix-vector multiplication on a hypercube with block 1-D partitioning:

Equation 8.3

$$T_o = t_s p \log p + t_w n p.$$

Recall from Chapter 5 that the central relation that determines the isoefficiency function of a parallel algorithm is W = KT_o (Equation 5.14), where K = E/(1 - E) and E is the desired efficiency. Rewriting this relation for matrix-vector multiplication, first with only the t_s term of To,

Equation 8.4

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

Equation 8.4 gives the isoefficiency term with respect to message startup time. Similarly, for the tw term of the overhead function,

$$W = K t_w n p.$$

Since W = n2 (Equation 8.1), we derive an expression for W in terms of p, K , and tw (that is, the isoefficiency function due to tw) as follows:

Equation 8.5

$$\begin{aligned} n^2 &= K t_w n p, \\ n &= K t_w p, \\ n^2 &= K^2 t_w^2 p^2, \\ W &= K^2 t_w^2 p^2. \end{aligned}$$

Now consider the degree of concurrency of this parallel algorithm. Using 1-D partitioning, a maximum of n processes can be used to multiply an n x n matrix with an n x 1 vector. In other words, p is O(n), which yields the following condition:

Equation 8.6

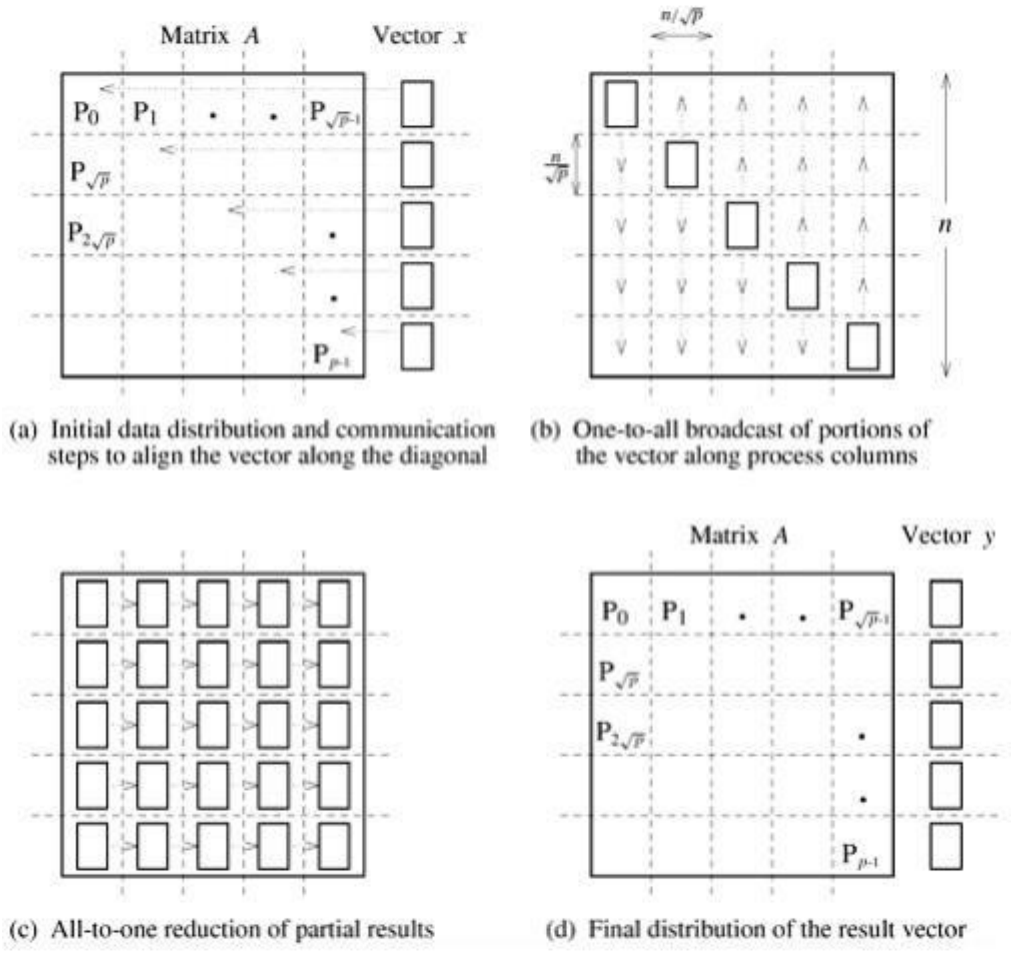
$$\begin{aligned} n &= \Omega(p), \\ n^2 &= \Omega(p^2), \\ W &= \Omega(p^2). \end{aligned}$$

The overall asymptotic isoefficiency function can be determined by comparing Equations 8.4, 8.5, and 8.6. Among the three, Equations 8.5 and 8.6 give the highest asymptotic rate at which the problem size must increase with the number of processes to maintain a fixed efficiency. This rate of Q(p2) is the asymptotic isoefficiency function of the parallel matrix-vector multiplication algorithm with 1-D partitioning.

2-D Partitioning

This section discusses parallel matrix-vector multiplication for the case in which the matrix is distributed among the processes using a block 2-D partitioning. Figure 8.2 shows the distribution of the matrix and the distribution and movement of vectors among the processes.

Figure 8.2. Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, p = n^2 if the matrix size is n x n.


Matrix A
Vector x
P_0
P_1
$P_{\sqrt{p}-1}$
$P_{\sqrt{p}}$
$P_{2\sqrt{p}}$
P_{p-1}
(a) Initial data distribution and communication steps to align the vector along the diagonal
$n/\sqrt{p}$
$\frac{n}{\sqrt{p}}$
n
(b) One-to-all broadcast of portions of the vector along process columns
(c) All-to-one reduction of partial results
Matrix A
Vector y
P_0
P_1
$P_{\sqrt{p}-1}$
$P_{\sqrt{p}}$
$P_{2\sqrt{p}}$
P_{p-1}
(d) Final distribution of the result vector

One Element Per Process:

We start with the simple case in which an n x n matrix is partitioned among n^2 processes such that each process owns a single element. The n x 1 vector x is distributed only in the last column of n processes, each of which owns one element of the vector. Since the algorithm multiplies the elements of the vector x with the corresponding elements in each row of the matrix, the vector must be distributed such that the i^{th} element of the vector is available to the i^{th} element of each row of the matrix. The communication steps for this are shown in Figure 8.2(a) and (b). Notice the similarity of Figure 8.2 to Figure 8.1. Before the multiplication, the elements of the matrix and the vector must be in the same relative locations as in Figure 8.1(c). However, the vector communication steps differ between various partitioning strategies. With 1- D partitioning, the elements of the vector cross only the horizontal partition-boundaries (Figure 8.1), but for 2-D partitioning, the vector elements cross both horizontal and vertical partition boundaries (Figure 8.2). As Figure 8.2(a) shows, the first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix. Often, the vector is stored along the diagonal instead of the last column, in which case this step is not required. The second step copies the vector elements from each diagonal process to all the processes in the corresponding column. As Figure 8.2(b) shows, this step consists of n simultaneous one-to-all broadcast operations, one in each column of processes. After these two communication steps, each process multiplies its matrix element with the corresponding element of x. To obtain the result vector y, the products computed for each row must be added, leaving the sums in the last column of processes. Figure 8.2(c) shows this step, which requires an all-to-one reduction (Section 4.1) in each row with the last process of the row as the destination. The parallel matrix-vector multiplication is complete after the reduction step. Parallel Run Time 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 time Q(log n). Since each process performs a single multiplication in constant time, the overall parallel run time of this algorithm is Q(n). The cost (process-time product) is Q(n^2 log n); hence, the algorithm is not cost-optimal.

Using Fewer than n^2 Processes:

A cost-optimal parallel implementation of matrix-vector multiplication with block 2-D partitioning of the matrix can be obtained if the granularity of computation at each process is increased by using fewer than n^2 processes. Consider a logical two-dimensional mesh of p processes in which 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. Figure 8.2 also illustrates the initial data-mapping and the various communication steps for this case. The entire vector must be distributed on each row of processes before the multiplication can be performed. First, the vector is aligned along the main diagonal. For this, each process in the rightmost column sends its $n/\sqrt{p}$ vector elements to the diagonal process in its row. Then a column wise one-to-all broadcast of these $n/\sqrt{p}$ elements takes place. Each process then performs n^2/p multiplications and locally adds the $n/\sqrt{p}$ sets of products. At the end of this step, as shown

$n/\sqrt{p}$

in Figure 8.2(c), each process has partial sums that must be accumulated along each row to obtain the result vector. Hence, the last step of the algorithm is an all-to-one reduction of the $n/\sqrt{p}$ values in each row, with the rightmost process of the row as the destination.

Parallel Run Time The first step of sending a message of size $n/\sqrt{p}$ from the rightmost process of a row to the diagonal process (Figure 8.2(a)) takes time $t_s + t_w n/\sqrt{p}$. . We can perform $(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$ the column wise one-to-all broadcast in at most time by using the procedure described in Section 4.1.3. Ignoring the time to perform additions, the final row wise all-to-one reduction also takes the same amount of time. Assuming that a multiplication and addition pair takes unit time, each process spends approximately n^2/p time in computation. Thus, the parallel run time for this procedure is as follows:

Equation 8.7

$$\begin{aligned} T_P &= \overbrace{n^2/p}^{\text{computation}} + \overbrace{t_s + t_w n/\sqrt{p}}^{\text{aligning the vector}} + \\ &\quad \overbrace{(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})}^{\text{columnwise one-to-all broadcast}} + \overbrace{(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})}^{\text{all-to-one reduction}} \\ &\approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p \end{aligned}$$

Scalability Analysis By using Equations 8.1 and 8.7, and applying the relation $T_o = pT_P - W$ (Equation 5.1), we get the following expression for the overhead function of this parallel algorithm:

Equation 8.8

$$T_o = t_s p \log p + t_w n \sqrt{p} \log p.$$

We now perform an approximate isoefficiency analysis along the lines of Section 5.4.2 by considering the terms of the overhead function one at a time (see Problem 8.4 for a more precise isoefficiency analysis). For the ts term of the overhead function, Equation 5.14 yields

Equation 8.9

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

Equation 8.9 gives the isoefficiency term with respect to the message startup time. We can obtain the isoefficiency function due to t_w by balancing the term $t_w n \sqrt{p}$ log p with the problem size n^2. Using the isoefficiency relation of Equation 5.14, we get the following:

Equation 8.10

$$\begin{aligned} W = n^2 &= K t_w n \sqrt{p} \log p, \\ n &= K t_w \sqrt{p} \log p, \\ n^2 &= K^2 t_w^2 p \log^2 p, \\ W &= K^2 t_w^2 p \log^2 p. \end{aligned}$$

Finally, considering that the degree of concurrency of 2-D partitioning is n^2 (that is, a maximum of n2 processes can be used), we arrive at the following relation:

Equation 8.11

$$\begin{aligned} p &= O(n^2), \\ n^2 &= \Omega(p), \\ W &= \Omega(p). \end{aligned}$$

Among Equations 8.9, 8.10, and 8.11, the one with the largest right-hand side expression determines the overall isoefficiency function of this parallel algorithm. To simplify the analysis, we ignore the impact of the constants and consider only the asymptotic rate of the growth of problem size that is necessary to maintain constant efficiency. The asymptotic isoefficiency term due to tw (Equation 8.10) clearly dominates the ones due to ts (Equation 8.9) and due to concurrency (Equation 8.11). Therefore, the overall asymptotic isoefficiency function is given by Q(p $\log^2$ p). The isoefficiency function also determines the criterion for cost-optimality (Section 5.4.3). With an isoefficiency function of Q(p log2 p), the maximum number of processes that can be used
cost-optimally for a given problem size W is determined by the following relations:

Equation 8.12

$$\begin{aligned} p \log^2 p &= O(n^2), \\ \log p + 2 \log\log p &= O(\log n). \end{aligned}$$

Ignoring the lower-order terms,

$$\log p = O(\log n).$$

Substituting log *n* for log *p* in Equation 8.12,

Equation 8.13

$$\begin{aligned} p \log^2 n &= O(n^2), \\ p &= O\left(\frac{n^2}{\log^2 n}\right). \end{aligned}$$

The right-hand side of Equation 8.13 gives an asymptotic upper bound on the number of processes that can be used cost-optimally for an n x n matrix-vector multiplication with a 2-D partitioning of the matrix. Comparison of 1-D and 2-D Partitionings A comparison of Equations 8.2 and 8.7 shows that matrix-vector multiplication is faster with block 2-D partitioning of the matrix than with block 1-D partitioning for the same number of processes. If the number of processes is greater than n, then the 1-D partitioning cannot be used. However, even if the number of processes is less than or equal to n, the analysis in this section suggests that 2-D partitioning is preferable. Among the two partitioning schemes, 2-D partitioning has a better (smaller) asymptotic isoefficiency function. Thus, matrix-vector multiplication is more scalable with 2-D partitioning; that is, it can deliver the same efficiency on more processes with 2-D partitioning than with 1-D partitioning.

Matrix-Matrix Multiplication:

This section discusses parallel algorithms for multiplying two n x n dense, square matrices A and B to yield the product matrix C = A x B. All parallel matrix multiplication algorithms in this chapter are based on the conventional serial algorithm shown in Algorithm 8.2. If we assume that an addition and multiplication pair (line 8) takes unit time, then the sequential run time of this algorithm is n^3. Matrix multiplication algorithms with better asymptotic sequential complexities are available, for example Strassen's algorithm. However, for the sake of simplicity, in this book we assume that the conventional algorithm is the best available serial algorithm. Problem 8.5 explores the performance of parallel matrix multiplication regarding Strassen's method as the base algorithm.

Algorithm 8.2 The conventional serial algorithm for multiplication of two n x n matrices.

```
1.   procedure MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to n - 1 do
4.          for j := 0 to n - 1 do
5.              begin
6.                 C[i, j] := 0;
7.                 for k := 0 to n - 1 do
8.                     C[i, j] := C[i, j] + A[i, k] x B[k, j];
9.              endfor;
10.  end MAT_MULT
```

Algorithm 8.3 The block matrix multiplication algorithm for n x n matrices with a block size of (n/q) x (n/q).

```
procedure BLOCK_MAT_MULT (A, B, C)
begin
   for i := 0 to q - 1 do
       for j := 0 to q - 1 do
           begin
              Initialize all elements of $C_{i,j}$ to zero;
              for k := 0 to q - 1 do
                  $C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j}$;
           endfor;
end BLOCK_MAT_MULT
```

A concept that is useful in matrix multiplication as well as in a variety of other matrix algorithms is that of block matrix operations. We can often express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix. Such algebraic operations on the submatrices are called block matrix operations. For example, an n x n matrix A can be regarded as a q x q array of blocks Ai,j (0 i, j < q) such that each block is an (n/q) x (n/q) submatrix. The matrix multiplication algorithm in Algorithm 8.2 can then be rewritten as Algorithm 8.3, in which the multiplication and addition operations on line 8 are matrix multiplication and matrix addition, respectively. Not only are the final results of Algorithm 8.2 and 8.3 identical, but so are the total numbers of scalar additions and multiplications performed by each. Algorithm 8.2 performs n^3 additions and multiplications, and Algorithm 8.3 performs q^3 matrix multiplications, each involving (n/q) x (n/q) matrices and requiring (n/q)3 additions and multiplications. We can use p processes to implement the block version of matrix multiplication in parallel by choosing $q = \sqrt{p}$ and computing a distinct $C_{i,j}$ block at each process. In the following sections, we describe a few ways of parallelizing Algorithm 8.3. Each of the following parallel matrix multiplication algorithms uses a block 2-D partitioning of the matrices.

A Simple Parallel Algorithm

Consider two n x n matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ of $0 \le k < \sqrt{p}$ size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes. The processes are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$. 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 $(0 \le i, j < \sqrt{p})$. To acquire all the required blocks, an all-to-all broadcast of matrix A's blocks is performed in each row of processes, and an all-to-all broadcast of matrix B's blocks is performed in each column. After $P_{i,j}$ acquires $A_{i,0}, A_{i,1}, \dots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \dots, B_{\sqrt{p}-1,j}$. it performs the submatrix multiplication and addition step of lines 7 and 8 in Algorithm 8.3.

Performance and Scalability Analysis The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the process mesh) among groups of $\sqrt{p}$ processes. The messages consist of submatrices of n^2/p elements. From Table 4.1, the total communication time is $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$. After the communication step, each process computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices (lines 7 and 8 of Algorithm 8.3 with $q = \sqrt{p}$. . This takes a total of time $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$. Thus, the parallel run time is approximately

Equation 8.14

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

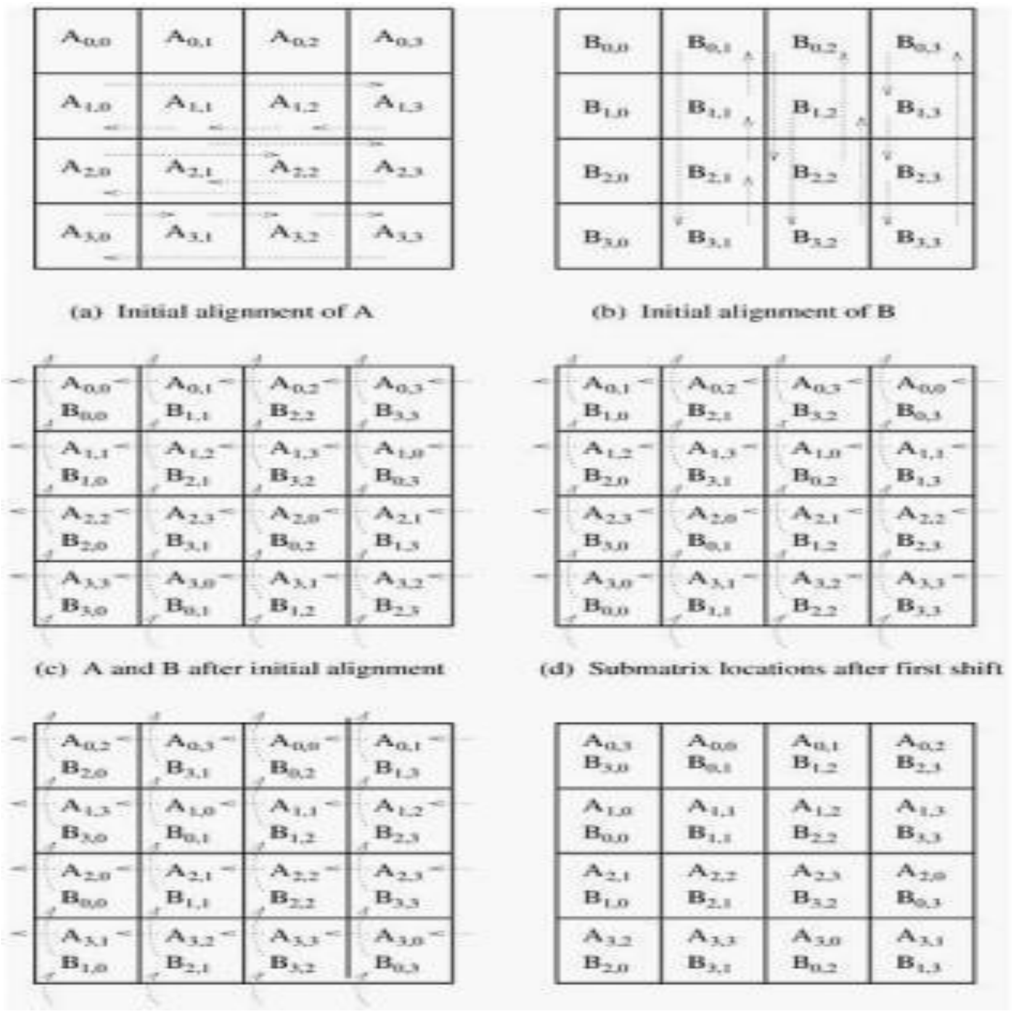
The process-time product is $n^3 + t_s p \log p + 2t_w n^2 \sqrt{p}$, and the parallel algorithm is cost-optimal for p = O(n^2).

The isoefficiency functions due to t_s and t_w are t_s p log p and 8$(t_w)^3$p3/2, respectively. Hence, the overall isoefficiency function due to the communication overhead is Q(p3/2). This algorithm can use a maximum of n^2 processes; hence, $p \leq n^2$ or $n^3 \geq p^{3/2}$. Therefore, the isoefficiency function due to concurrency is also Q(p3/2). A notable drawback of this algorithm is its excessive memory requirements. At the end of the communication phase, each process has $\sqrt{p}$ blocks of both matrices A and B. Since each block requires Q(n^2/p) memory, each process requires $\Theta(n^2/\sqrt{p})$ memory. The total memory requirement over all the processes is $\Theta(n^2/\sqrt{p})$, which is $\sqrt{p}$ times the memory requirement of the sequential algorithm.

Cannon's Algorithm:

Cannon's algorithm is a memory-efficient version of the simple algorithm presented in Section 8.2.1. To study this algorithm, we again partition matrices A and B into p square blocks. We label the processes from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$, and initially assign submatrices $A_{i,j}$ and $B_{i,j}$ to process $P_{i,j}$. Although every process in the i th row requires all submatrices $A_{i,k}$ $(0 \leq k < \sqrt{p})$, it is possible to schedule the computations of the processes of the ith row such that, at any given time, each process is using a different $A_{i,k}$. 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. If an identical schedule is applied to the columns, then no process holds more than one block of each matrix at any time, and the total memory requirement of the algorithm over all the processes is Q(n2). Cannon's algorithm is based on this idea. The scheduling for the multiplication of submatrices on separate processes in Cannon's algorithm is illustrated in Figure 8.3 for 16 processes.

Figure 8.3. The communication steps in Cannon's algorithm on 16 processes.


(a) Initial alignment of A (b) Initial alignment of B

(c) A and B after initial alignment (d) Submatrix locations after first shift

The first communication step of the algorithm aligns the blocks of A and B in such a way that each process multiplies its local submatrices. As Figure 8.3(a) shows, this alignment is achieved for matrix A by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by i steps. Similarly, as shown in Figure 8.3(b), all submatrices $B_{i,j}$ are shifted up (with wraparound) by j steps. These are circular shift operations (Section 4.6) in each row and column of processes, which leave process Pi,j with submatrices and $A_{i,(j+i)\bmod\sqrt{p}}$ and $B_{(i+j)\bmod\sqrt{p},j}$. Figure 8.3(c) shows the blocks of A and B after the initial alignment, when each process is ready for the first submatrix multiplication. After a submatrix multiplication step, each block of A moves one step left and each block of B moves one step up (again with wraparound), as shown in Figure 8.3(d). A sequence of $\sqrt{p}$ such submatrix multiplications and single-step shifts pairs up each $A_{i,k}$ and $B_{k,j}$ for $k \ (0 \leq k < \sqrt{p})$ at $P_{i,j}$. This completes the multiplication of matrices A and B.

Performance Analysis The initial alignment of the two matrices (Figure 8.3(a) and (b)) involves a rowwise and a columnwise circular shift. In any of these shifts, the maximum distance over which a block shifts is $\sqrt{p} - 1$. The two shift operations require a total of time 2(ts + twn2/p) (Table 4.1). Each of the single-step shifts in the compute-and-shift phase of the algorithm takes time $t_s + t_w n^2/p$. Thus, the total communication time (for both matrices) during this phase of the algorithm is $2(t_s + t_w n^2/p)\sqrt{p}$. For large enough p on a network with sufficient bandwidth, the communication time for the initial alignment can be disregarded in comparison with the time spent in communication during the compute-and-shift phase. Each process performs multiplications of submatrices. Assuming that a multiplication and addition pair takes unit time, the total time that each process spends in computation is n3/p. Thus, the approximate overall parallel run time of this algorithm is

Equation 8.15

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}.$$

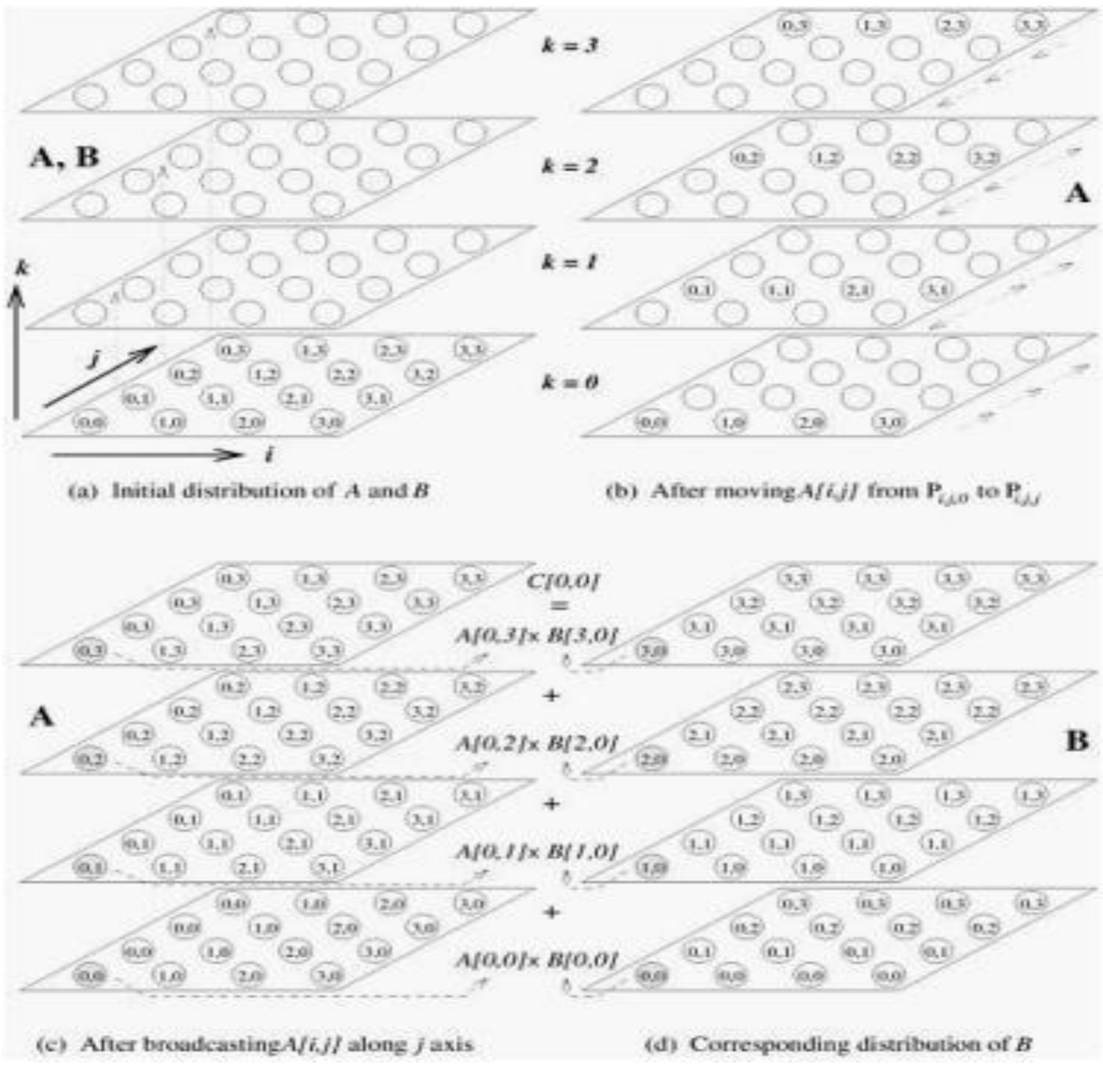
The cost-optimality condition for Cannon's algorithm is identical to that for the simple algorithm presented in Section 8.2.1. As in the simple algorithm, the isoefficiency function of Cannon's algorithm is Q($p^{3/2}$).

The DNS Algorithm

The matrix multiplication algorithms presented so far use block 2-D partitioning of the input and the output matrices and use a maximum of n^2 processes for n x n matrices. As a result, these algorithms have a parallel run time of W(n) because there are Q(n^3) operations in the serial algorithm. We now present a parallel algorithm based on partitioning intermediate data that can use up to n^3 processes and that performs matrix multiplication in time Q(log n) by using W(n^3/log n) processes. This algorithm is known as the DNS algorithm because it is due to Dekel, Nassimi, and Sahni. We first introduce the basic idea, without concern for inter-process communication. Assume that n^3 processes are available for multiplying two n x n matrices. These processes are arranged in a three-dimensional n x n x n logical array. Since the matrix multiplication algorithm performs n3 scalar multiplications, each of the n3 processes is assigned a single scalar multiplication. The processes are labeled according to their location in the array, and the multiplication A[i, k] x B[k, j] is assigned to process $P_{i,j,k}$ $(0 \leq i, j, k < n)$ After each process performs a single multiplication, the contents of Pi,j,0, Pi,j,1, ..., Pi,j,n-1 are added to obtain C [i, j]. The additions for all C [i, j] can be carried out simultaneously in log n steps each. Thus, it takes one step to multiply and log n steps to add; that is, it takes time Q(log n) to multiply the n x n matrices by this algorithm. We now describe a practical parallel implementation of matrix multiplication based on this idea. As Figure 8.4 shows, the process arrangement can be visualized as n planes of n x n processeseach. Each plane corresponds to a different value of k. Initially, as shown in Figure 8.4(a), the matrices are distributed among the n2 processes of the plane corresponding to k = 0 at the base of the three-dimensional process array. Process Pi,j,0 initially owns A[i, j] and B[i, j].

Figure 8.4. The communication steps in the DNS algorithm while multiplying 4 x 4 matrices A and B on

64 processes. The shaded processes in part (c) store elements of the first row of A and the shaded processes in part (d) store elements of the first column of B.



(a) Initial distribution of A and B (b) After moving A[i,j] from $P_{i,j,0}$ to $P_{i,j,j}$

(c) After broadcasting A[i,j] along j axis (d) Corresponding distribution of B

The vertical column of processes Pi,j,* computes the dot product of row A[i, *] and column B[*, j]. Therefore, rows of A and columns of B need to be moved appropriately so that each vertical column of processes $P_{i,j,}$* has row A[i, *] and column B[*, j]. More precisely, process Pi,j,k should have A[i, k] and B[k, j]. The communication pattern for distributing the elements of matrix A among the processes is shown in Figure 8.4(a)–(c). First, each column of A moves to a different plane such that the j th column occupies the same position in the plane corresponding to k = j as it initially did in the plane corresponding to k = 0. The distribution of A after moving A[i, j] from Pi,j,0 to Pi,j,j is shown in Figure 8.4(b). Now all the columns of A are replicated n times in their respective planes by a parallel one-to-all broadcast along the j axis. The result of this step is shown in Figure 8.4(c), in which the n processes Pi,0,j, Pi,1,j, ..., Pi,n-1,j receive a copy of A[i, j] from Pi,j,j. At this point, each vertical column of processes Pi,j,* has row A[i, *]. More precisely, process Pi,j,k has A[i, k]. For matrix B, the communication steps are similar, but the roles of i and j in process subscripts are switched. In the first one-to-one communication step, B[i, j] is moved from Pi,j,0 to Pi,j,i. Then it is broadcast from Pi,j,i among P0,j,i, P1,j,i, ..., Pn-1,j,i. The distribution of B after this onetoall broadcast along the i axis is shown in Figure 8.4(d). At this point, each vertical column of processes Pi,j,* has column B[*, j]. Now process Pi,j,k has B[k, j], in addition to A[i, k]. After these communication steps, A[i, k] and B[k, j] are multiplied at Pi,j,k. Now each element C[i, j] of the product matrix is obtained by an all-to-one reduction along the k axis. During this step, process Pi,j,0 accumulates the results of the multiplication from processes Pi,j,1, ..., Pi,j,n-1. Figure 8.4 shows this step for C[0, 0]. The DNS algorithm has three main communication steps: (1) moving the columns of A and the rows of B to their respective planes, (2) performing one-to-all broadcast along the j axis for A and along the i axis for B, and (3) all-to-one reduction along the k axis. All these operations are performed within groups of n processes and take time Q(log n). Thus, the parallel run time for multiplying two n x n matrices using the DNS algorithm on n3 processes is Q(log n).

DNS Algorithm with Fewer than n ^{3} Processes:

The DNS algorithm is not cost-optimal for n^3 processes, since its process-time product of Q(n^3 log n) exceeds the Q(n^3) sequential complexity of matrix multiplication. We now present a costoptimal version of this algorithm that uses fewer than n^3 processes. Another variant of the DNS algorithm that uses fewer than n3 processes is described in Problem 8.6.

Assume that the number of processes p is equal to q3 for some q < n. To implement the DNS algorithm, the two matrices are partitioned into blocks of size (n/q) x (n/q). Each matrix can thus be regarded as a q x q two-dimensional square array of blocks. The implementation of this algorithm on q3 processes is very similar to that on n3 processes. The only difference is that now we operate on blocks rather than on individual elements. Since $1 \leq q \leq n,$, the number of processes can vary between 1 and n3. **Performance Analysis** The first one-to-one communication step is performed for both A and B, and takes time ts + tw(n/q)2 for each matrix. The second step of one-to-all broadcast is also performed for both matrices and takes time ts log q + tw(n/q)2 log q for each matrix. The final all-to-one reduction is performed only once (for matrix C)and takes time ts log q + tw(n/q)2 log q. The multiplication of (n/q) x (n/q) submatrices by each process takes time (n/q) 3. We can ignore the communication time for the first one-to-one communication step because it is much smaller than the communication time of one-to-all broadcasts and all-to-one reduction. We can also ignore the computation time for addition in the final reduction phase because it is of a smaller order of magnitude than the computation time for multiplying the submatrices. With these assumptions, we get the following approximate expression for the parallel run time of the DNS algorithm:

$$T_P \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

Since $q = p^{1/3}$, we get

Equation 8.16

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$$

The total cost of this parallel algorithm is $n^3 + t_s\, p \log p + t_w n^2 p^{1/3} \log p$. The isoefficiency function is $\Theta(p(\log p)^3)$. The algorithm is cost-optimal for $n^3 = \Omega(p(\log p)^3)$, or $p = O(n^3/(\log n)^3)$.

Issues in Sorting on Parallel Computers

Parallelizing a sequential sorting algorithm involves distributing the elements to be sorted onto the available processes. This process raises a number of issues that we must address in order to make the presentation of parallel sorting algorithms clearer.

Where the Input and Output Sequences are Stored

In sequential sorting algorithms, the input and the sorted sequences are stored in the process's memory. However, in parallel sorting there are two places where these sequences can reside. They may be stored on only one of the processes, or they may be distributed among the processes. The latter approach is particularly useful if sorting is an intermediate step in another algorithm. In this chapter, we assume that the input and sorted sequences are distributed among the processes.

Consider the precise distribution of the sorted output sequence among the processes. A general method of distribution is to enumerate the processes and use this enumeration to specify a global ordering for the sorted sequence. In other words, the sequence will be sorted with respect to this process enumeration. For instance, if P_i comes before P_j in the enumeration, all the elements stored in P_i will be smaller than those stored in P_j . We can enumerate the processes in many ways. For certain parallel algorithms and interconnection networks, some enumerations lead to more efficient parallel formulations than others.

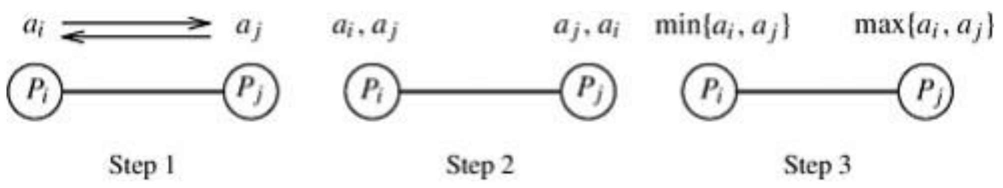
How Comparisons are Performed

A sequential sorting algorithm can easily perform a compare-exchange on two elements because they are stored locally in the process's memory. In parallel sorting algorithms, this step is not so easy. If the elements reside on the same process, the comparison can be done easily. But if the elements reside on different processes, the situation becomes more complicated.

One Element Per Process

Consider the case in which each process holds only one element of the sequence to be sorted. At some point in the execution of the algorithm, a pair of processes (Pi, Pj) may need to compare their elements, ai and aj. After the comparison, Pi will hold the smaller and Pj the larger of {ai, aj}. We can perform comparison by having both processes send their elements to each other. Each process compares the received element with its own and retains the appropriate element. In our example, Pi will keep the smaller and Pj will keep the larger of {ai, aj}. As in the sequential case, we refer to this operation as compare-exchange. As Figure 9.1 illustrates, each compare-exchange operation requires one comparison step and one communication step.

A parallel compare-exchange operation. Processes Pi and Pj send their elements to each other. Process Pi keeps min{ai, aj}, and Pj keeps max{ai , aj}.



If we assume that processes Pi and Pj are neighbors, and the communication channels are bidirectional, then the communication cost of a compare-exchange step is (ts + tw), where ts and tw are message-startup time and per-word transfer time, respectively. In commercially available message-passing computers, ts is significantly larger than tw, so the communication time is dominated by ts. Note that in today's parallel computers it takes more time to send an element from one process to another than it takes to compare the elements. Consequently, any parallel sorting formulation that uses as many processes as elements to be sorted will deliver very poor performance because the overall parallel run time will be dominated by inter process communication.

More than One Element Per Process:

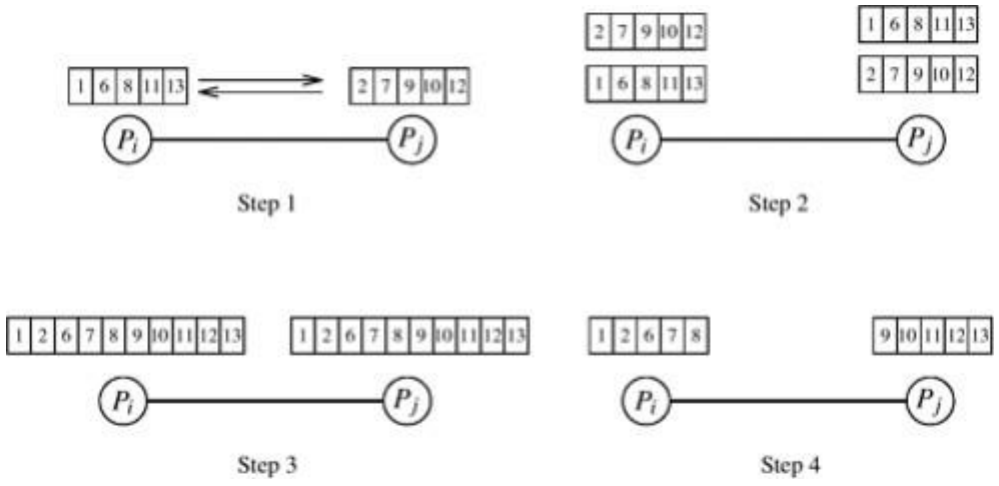
A general-purpose parallel sorting algorithm must be able to sort a large sequence with a relatively small number of processes. Let p be the number of processes P0, P1, ..., Pp-1, and let n be the number of elements to be sorted. Each process is assigned a block of n/p elements, and all the processes cooperate to sort the sequence. Let A0, A1, ... A p-1 be the blocks assigned to processes P0, P1, ... Pp-1, respectively. We say that Ai Aj if every element of Ai is less than or equal to every element in Aj. When the sorting algorithm finishes, each process Pi holds a set such that

$$A'_i \leq A'_j \text{ for } i \leq j, \text{ and } \bigcup_{i=0}^{p-1} A_i = \bigcup_{i=0}^{p-1} A'_i.$$

As in the one-element-per-process case, two processes Pi and Pj may have to redistribute their blocks of n/p elements so that one of them will get the smaller n/p elements and the other will get the larger n/p elements. Let Ai and Aj be the blocks stored in processes Pi and Pj. If the block of n/p elements at each process is already sorted, the redistribution can be done efficiently as follows. Each process sends its block to the other process. Now, each process merges the two sorted blocks and retains only the appropriate half of the merged block. We refer to this operation of comparing and splitting two sorted blocks as compare-split. The compare-split operation is illustrated in Figure 9.2.

Figure 9.2. A compare-split operation. Each process sends its block of size n/p to the other process.

Each process merges the received block with its own block and retains only the appropriate half of the merged block. In this example, process Pi retains the smaller elements and process Pj retains the larger elements.



If we assume that processes Pi and Pj are neighbors and that the communication channels are bidirectional, then the communication cost of a compare-split operation is (ts + twn/p). As the block size increases, the significance of ts decreases, and for sufficiently large blocks it can be ignored. Thus, the time required to merge two sorted blocks of n/p elements is Q(n/p).

Bubble Sort and its Variants:

The previous section presented a sorting network that could sort n elements in a time of Q(log2 n). We now turn our attention to more traditional sorting algorithms. Since serial algorithms with Q(n log n) time complexity exist, we should be able to use Q(n) processes to sort n elements in time Q(log n). As we will see, this is difficult to achieve. We can, however, easily parallelize many sequential sorting algorithms that have Q(n2) complexity. The algorithms we present are based on bubble sort. The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted. Given a sequence , the algorithm first performs n - 1 compare-exchange operations in the following order: (a1, a2), (a2, a3), ..., (an-1, an). This step moves the largest element to the end of the sequence. The last element in the transformed sequence is then ignored, and the sequence of compare-exchanges is applied to the resulting sequence . The sequence is sorted after n - 1 iterations. We can improve the performance of bubble sort by terminating when no exchanges take place during an iteration. The bubble sort algorithm is shown in Algorithm 9.2. An iteration of the inner loop of bubble sort takes time Q(n), and we perform a total of Q(n) iterations; thus, the complexity of bubble sort is Q(n2). Bubble sort is difficult to parallelize. To see this, consider how compare-exchange operations are performed during each phase of the algorithm (lines 4 and 5 of Algorithm 9.2).

Bubble sort compares all adjacent pairs in order; hence, it is inherently sequential. In the following two sections, we present two variants of bubble sort that are well suited to parallelization.

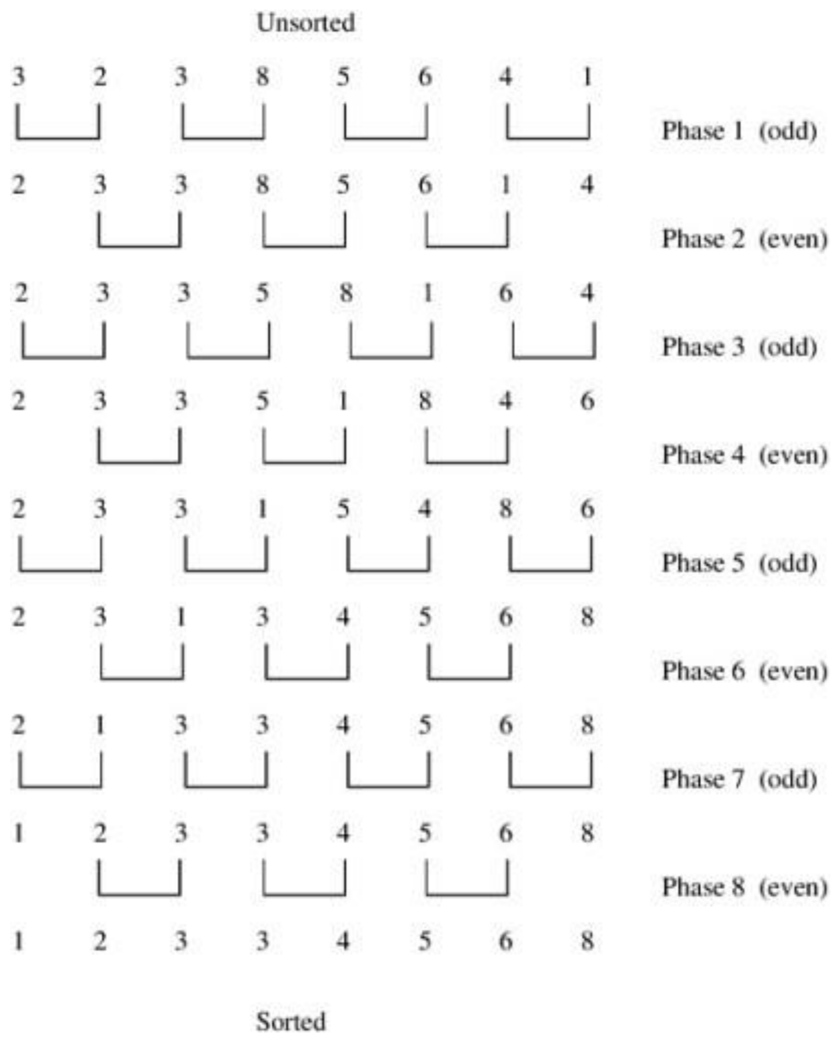
Algorithm 9.2 Sequential bubble sort algorithm.

```
procedure BUBBLE_SORT($n$)
begin
   for $i$ := $n$ - 1 downto 1 do
       for $j$ := 1 to $i$ do
           compare-exchange($a_j$, $a_{j+1}$);
end BUBBLE_SORT
```

Odd-Even Transposition

The odd-even transposition algorithm sorts n elements in n phases (n is even), each of which requires n/2 compare-exchange operations. This algorithm alternates between two phases, called the odd and even phases. Let be the sequence to be sorted. During the odd phase, elements with odd indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a1, a2), (a3, a4), ..., (an-1, an) are compare-exchanged (assuming n is even). Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (a2, a3), (a4, a5), ..., (an-2, an-1) are compare-exchanged. After n phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires Q(n) comparisons, and there are a total of n phases; thus, the sequential complexity is Q(n2). The odd-even transposition sort is shown in Algorithm 9.3 and is illustrated in Figure 9.13.

Figure 9.13. Sorting n = 8 elements, using the odd-even transposition sort algorithm. During each phase, n = 8 elements are compared.



Algorithm 9.3 Sequential odd-even transposition sort algorithm.

```
procedure ODD-EVEN($n$)
begin
   for $i$ := 1 to $n$ do
   begin
      if $i$ is odd then
           for $j$ := 0 to $n/2 - 1$ do
               compare-exchange($a_{2j+1}$, $a_{2j+2}$);
      if $i$ is even then
           for $j$ := 1 to $n/2 - 1$ do
               compare-exchange($a_{2j}$, $a_{2j+1}$);
   end for
end ODD-EVEN
```

Parallel Formulation

It is easy to parallelize odd-even transposition sort. During each phase of the algorithm, compare-exchange operations on pairs of elements are performed simultaneously. Consider the one-element-per-process case. Let n be the number of processes (also the number of elements to be sorted). Assume that the processes are arranged in a one-dimensional array. Element ai initially resides on process Pi for i = 1, 2, ..., n. During the odd phase, each process that has an odd label compare-exchanges its element with the element residing on its right neighbor. Similarly, during the even phase, each process with an even label compare-exchanges its element with the element of its right neighbor. This parallel formulation is presented in Algorithm 9.4.

Algorithm 9.4 The parallel formulation of odd-even transposition sort on an n-process ring.

```
procedure ODD-EVEN_PAR (n)
begin
   id := process's label
   for i := 1 to n do
   begin
      if i is odd then
          if id is odd then
              compare-exchange_min(id + 1);
          else
              compare-exchange_max(id - 1);
      if i is even then
          if id is even then
              compare-exchange_min(id + 1);
          else
              compare-exchange_max(id - 1);
   end for
end ODD-EVEN_PAR
```

During each phase of the algorithm, the odd or even processes perform a compare- exchange step with their right neighbors. As we know from Section 9.1, this requires time Q(1). A total of n such phases are performed; thus, the parallel run time of this formulation is Q(n). Since the sequential complexity of the best sorting algorithm for n elements is Q(n log n), this formulation of odd-even transposition sort is not cost-optimal, because its process-time product is Q(n2). To obtain a cost-optimal parallel formulation, we use fewer processes. Let p be the number of processes, where p < n. Initially, each process is assigned a block of n/p elements, which it sorts internally (using merge sort or quicksort) in Q((n/p) log(n/p)) time. After this, the processes execute p phases (p/2 odd and p/2 even), performing compare-split operations. At the end of these phases, the list is sorted (Problem 9.10). During each phase, Q(n/p) comparisons are performed to merge two blocks, and time Q(n/p) is spent communicating. Thus, the parallel run time of the formulation is

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

Since the sequential complexity of sorting is Q(n log n), the speedup and efficiency of this formulation are as follows:

Equation 9.6

$$S = \frac{\Theta(n\log n)}{\Theta((n/p)\log(n/p)) + \Theta(n)}$$

$$E = \frac{1}{1 - \Theta((\log p)/(\log n)) + \Theta(p/\log n)}$$

From Equation 9.6, odd-even transposition sort is cost-optimal when p = O(log n). The isoefficiency function of this parallel formulation is Q(p 2p), which is exponential. Thus, it is poorly scalable and is suited to only a small number of processes.

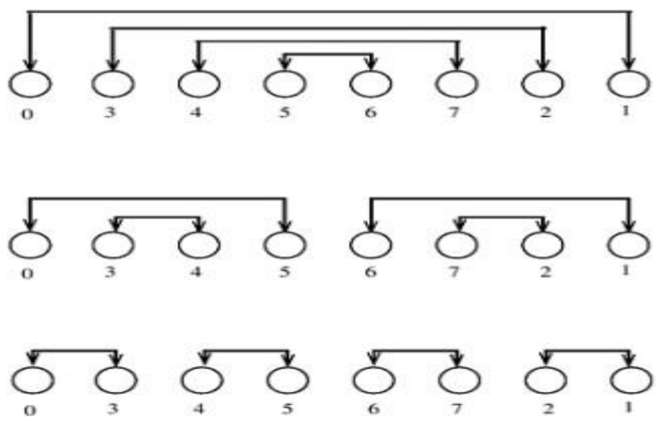
Shellsort:

The main limitation of odd-even transposition sort is that it moves elements only one position at a time. If a sequence has just a few elements out of order, and if they are Q(n) distance from their proper positions, then the sequential algorithm still requires time Q(n2) to sort the sequence. To make a substantial improvement over odd-even transposition sort, we need an algorithm that moves elements long distances. Shellsort is one such serial sorting algorithm.

Let n be the number of elements to be sorted and p be the number of processes. To simplify the presentation we will assume that the number of processes is a power of two, that is, p = 2d, but the algorithm can be easily extended to work for an arbitrary number of processes as well. Each process is assigned a block of n/p elements. The processes are considered to be arranged in a logical one-dimensional array, and the ordering of the processes in that array defines the global ordering of the sorted sequence. The algorithm consists of two phases. During the first phase, processes that are far away from each other in the array compare-split their elements. Elements thus move long distances to get close to their final destinations in a few steps. During the second phase, the algorithm switches to an odd-even transposition sort similar to the one described in the previous section. The only difference is that the odd and even phases are performed only as long as the blocks on the processes are changing. Because the first phase of the algorithm moves elements close to their final destinations, the number of odd and even phases performed by the second phase may be substantially smaller than p.

Initially, each process sorts its block of n/p elements internally in Q(n/p log(n/p)) time. Then, each process is paired with its corresponding process in the reverse order of the array. That is, process Pi, where i < p/2, is paired with process Pp-i -1. Each pair of processes performs a compare-split operation. Next, the processes are partitioned into two groups; one group has the first p/2 processes and the other group has the last p/2 processes. Now, each group is treated as a separate set of p/2 processes and the above scheme of process-pairing is applied to determine which processes will perform the compare-split operation. This process continues for d steps, until each group contains only a single process. The compare-split operations of the first phase are illustrated in Figure 9.14 for d = 3. Note that it is not a direct parallel formulation of the sequential shellsort, but it relies on similar ideas.

Figure 9.14. An example of the first phase of parallel shellsort on an eight-process array.



In the first phase of the algorithm, each process performs d = log p compare-split operations. In each compare-split operation a total of p/2 pairs of processes need to exchange their locally stored n/p elements. The communication time required by these compare-split operations depend on the bisection bandwidth of the network. In the case in which the bisection bandwidth is Q(p), the amount of time required by each operation is Q(n/p). Thus, the complexity of this phase is Q((n log p)/p). In the second phase, l odd and even phases are performed, each requiring time Q(n/p). Thus, the parallel run time of the algorithm is

Equation 9.7

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right)}^{\text{first phase}} + \overbrace{\Theta\left(l\frac{n}{p}\right)}^{\text{second phase}}.$$

The performance of shellsort depends on the value of l. If l is small, then the algorithm performs significantly better than odd-even transposition sort; if l is Q(p), then both algorithms perform similarly. Problem 9.13 investigates the worst-case value of l.

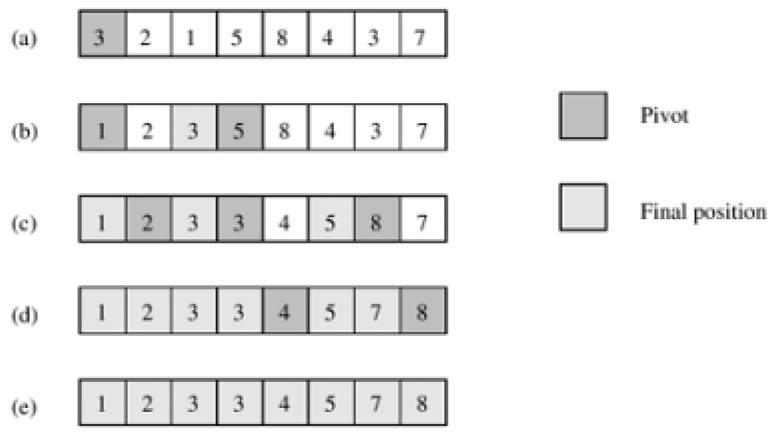
Quicksort

All the algorithms presented so far have worse sequential complexity than that of the lower bound for comparison-based sorting, Q(n log n). This section examines the quicksort algorithm, which has an average complexity of Q(n log n). Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.

Quicksort is a divide-and-conquer algorithm that sorts a sequence by recursively dividing it into smaller subsequences. Assume that the n-element sequence to be sorted is stored in the array A[1...n]. Quicksort consists of two steps: divide and conquer. During the divide step, a sequence A[q...r] is partitioned (rearranged) into two nonempty subsequences A[q...s] and A[s + 1...r] such that each element of the first subsequence is smaller than or equal to each element of the second subsequence. During the conquer step, the subsequences are sorted by recursively applying quicksort. Since the subsequences A[q...s] and A[s +

1...r] are sorted and the first subsequence has smaller elements than the second, the entire sequence is sorted. How is the sequence A[q...r] partitioned into two parts – one with all elements smaller than the other? This is usually accomplished by selecting one element x from A[q...r] and using this element to partition the sequence A[q...r] into two parts – one with elements less than or equal to x and the other with elements greater than x.Element x is called the pivot. The quicksort algorithm is presented in Algorithm 9.5. This algorithm arbitrarily chooses the first element of the sequence A[q...r] as the pivot. The operation of quicksort is illustrated in Figure 9.15.

Figure 9.15. Example of the quicksort algorithm sorting a sequence of size *n* = 8.



Algorithm 9.5 The sequential quicksort algorithm.

```
procedure QUICKSORT (A, q, r)
begin
   if q < r then
   begin
      x := A[q];
      s := q;
      for i := q + 1 to r do
          if A[i] ≤ x then
          begin
             s := s + 1;
             swap(A[s], A[i]);
          end if
      swap(A[q], A[s]);
      QUICKSORT (A, q, s);
      QUICKSORT (A, s + 1, r);
   end if
end QUICKSORT
```

The complexity of partitioning a sequence of size k is Q(k). Quicksort's performance is greatly affected by the way it partitions a sequence. Consider the case in which a sequence of size k is split poorly, into two subsequences of sizes 1 and k - 1. The run time in this case is given by the recurrence relation T(n) = T(n - 1) + Q(n), whose solution is T(n) = Q(n2). Alternatively, consider the case in which the sequence is split well, into two roughly equal-size subsequences of and elements. In this case, the run time is given by the recurrence relation T(n) = 2T(n/2) + Q(n), whose solution is T(n) = Q(n log n). The second split yields an optimal algorithm. Although quicksort can have O(n2) worst-case complexity, its average complexity is significantly better; the average number of compare-exchange operations needed by quicksort for sorting a randomly-ordered input sequence is 1.4n log n, which is asymptotically optimal. There are several ways to select pivots. For example, the pivot can be the median of a small number of elements of the sequence, or it can be an element selected at random. Some pivot selection strategies have advantages over others for certain input sequences.

Parallelizing Quicksort:

Quicksort can be parallelized in a variety of ways. First, consider a naive parallel formulation that was also discussed briefly in Section 3.2.1 in the context of recursive decomposition. Lines 14 and 15 of Algorithm 9.5 show that, during each call of QUICKSORT, the array is partitioned into two parts and each part is solved recursively. Sorting the smaller arrays represents two completely independent sub problems that can be solved in parallel. Therefore, one way to parallelize quicksort is to execute it initially on a single process; then, when the algorithm performs its recursive calls (lines 14 and 15), assign one of the sub problems to another process. Now each of these processes sorts its array by using quicksort and assigns one of its sub problems to other processes. The algorithm terminates when the arrays cannot be further partitioned. Upon termination, each process holds an element of the array, and the sorted order can be recovered by traversing the processes as we will describe later. This parallel formulation of quicksort uses n processes to sort n elements. Its major drawback is that partitioning the array A[q ...r] into two smaller arrays, A[q ...s] and A[s + 1 ...r], is done by a single process. Since one process must partition the original array A[1 ...n], the run time of this formulation is bounded below by W(n). This formulation is not cost-optimal, because its process-time product is W(n2).

The main limitation of the previous parallel formulation is that it performs the partitioning step serially. As we will see in subsequent formulations, performing partitioning in parallel is essential in obtaining an efficient parallel quicksort. To see why, consider the recurrence equation T (n) = 2T (n/2) + Q(n), which gives the complexity of quicksort for optimal pivot selection. The term Q(n) is due to the partitioning of the array. Compare this complexity with the overall complexity of the algorithm, Q(n log n). From these two complexities, we can think of the quicksort algorithm as consisting of Q(log n) steps, each requiring time Q(n) – that of splitting the array. Therefore, if the partitioning step is performed in time Q(1), using

Q(n) processes, it is possible to obtain an overall parallel run time of Q(log n), which leads to a cost optimal formulation. However, without parallelizing the partitioning step, the best we can do (while maintaining cost-optimality) is to use only Q(log n) processes to sort n elements in time Q(n) (Problem 9.14). Hence, parallelizing the partitioning step has the potential to yield a significantly faster parallel formulation.

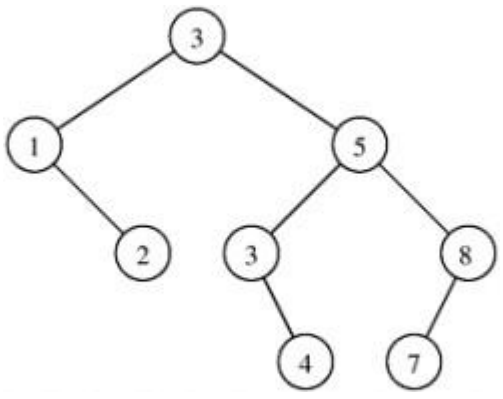
In the previous paragraph, we hinted that we could partition an array of size n into two smaller arrays in time Q(1) by using Q(n) processes. However, this is difficult for most parallel computing models. The only known algorithms are for the abstract PRAM models. Because of communication overhead, the partitioning step takes longer than Q(1) on realistic shared address-space and message-passing parallel computers. In the following sections we present three distinct parallel formulations: one for a CRCW PRAM, one for a shared-address-space architecture, and one for a message-passing platform. Each of these formulations parallelizes quicksort by performing the partitioning step in parallel.

Parallel Formulation for a CRCW PRAM

We will now present a parallel formulation of quicksort for sorting n elements on an n-process arbitrary CRCW PRAM. Recall from Section 2.4.1 that an arbitrary CRCW PRAM is a concurrent read, concurrent-write parallel random-access machine in which write conflicts are resolved arbitrarily. In other words, when more than one process tries to write to the same memory location, only one arbitrarily chosen process is allowed to write, and the remaining writes are ignored.

Executing quicksort can be visualized as constructing a binary tree. In this tree, the pivot is the root; elements smaller than or equal to the pivot go to the left subtree, and elements larger than the pivot go to the right subtree. Figure 9.16 illustrates the binary tree constructed by the execution of the quicksort algorithm illustrated in Figure 9.15. The sorted sequence can be obtained from this tree by performing an in-order traversal. The PRAM formulation is based on this interpretation of quicksort.

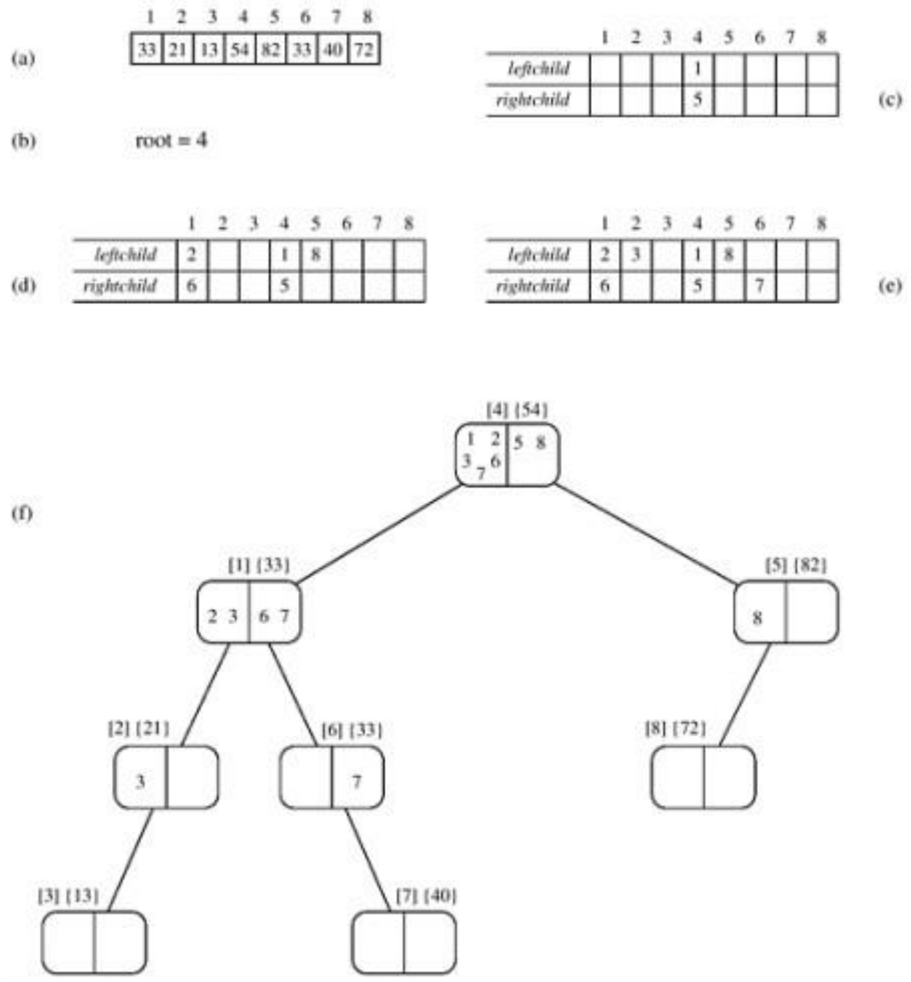
Figure 9.16. A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different arraypartitioning iteration. If pivot selection is optimal, then the height of the tree is Q(log n), which is also the number of iterations.



The algorithm starts by selecting a pivot element and partitioning the array into two parts – one with elements smaller than the pivot and the other with elements larger than the pivot. Subsequent pivot elements, one for each new subarray, are then selected in parallel. This formulation does not rearrange elements;

instead, since all the processes can read the pivot in constant time, they know which of the two subarrays (smaller or larger) the elements assigned to them belong to. Thus, they can proceed to the next iteration. The algorithm that constructs the binary tree is shown in Algorithm 9.6. The array to be sorted is stored in A[1...n] and process i is assigned element A[i]. The arrays leftchild[1...n] and rightchild[1...n] keep track of the children of a given pivot. For each process, the local variable parenti stores the label of the process whose element is the pivot. Initially, all the processes write their process labels into the variable root in line 5. Because the concurrent write operation is arbitrary, only one of these labels will actually be written into root. The value A[root] is used as the first pivot and root is copied into parenti for each process i . Next, processes that have elements smaller than A[parenti] write their process labels into left child[parenti], and those with larger elements write their process label into right child[parenti]. Thus, all processes whose elements belong in the smaller partition have written their labels into left child[parenti], and those with elements in the larger partition have written their labels into right child[parenti]. Because of the arbitrary concurrent-write operations, only two values – one for left child[parenti] and one for right child[parenti] – are written into these locations. These two values become the labels of the processes that hold the pivot elements for the next iteration, in which two smaller arrays are being partitioned. The algorithm continues until n pivot elements are selected. A process exits when its element becomes a pivot. The construction of the binary tree is illustrated in Figure 9.17. During each iteration of the algorithm, a level of the tree is constructed in time Q(1). Thus, the average complexity of the binary tree building algorithm is Q(log n) as the average height of the tree is Q(log n) (Problem 9.16).

Figure 9.17. The execution of the PRAM algorithm on the array shown in (a). The arrays left child and right child are shown in (c), (d), and (e) as the algorithm progresses. Figure (f) shows the binary tree constructed by the algorithm. Each node is labeled by the process (in square brackets), and the element is stored at that process (in curly brackets). The element is the pivot. In each node, processes with smaller elements than the pivot are grouped on the left side of the node, and those with larger elements are grouped on the right side. These two groups form the two partitions of the original array. For each partition, a pivot element is selected at random from the two groups that form the children of the node.



Algorithm 9.6 The binary tree construction procedure for the CRCW PRAM parallel quicksort formulation.

```
procedure BUILD TREE (A[1...n])
begin
   for each process i do
   begin
      root := i;
      parent_i := root;
      leftchild[i] := rightchild[i] := n + 1;
   end for
   repeat for each process i ≠ r oot do
   begin
      if (A[i] < A[parent_i]) or
           (A[i]= A[parent_i] and i <parent_i) then
      begin
         leftchild[parent_i] :=i ;
         if i = leftchild[parent_i] then exit
         else parent_i := leftchild[parent_i];
```

```
            end for
            else
            begin
               rightchild[$parent_i$] :=i;
               if i = rightchild[$parent_i$] then exit
               else $parent_i$ := rightchild[$parent_i$];
            end else
         end repeat
      end BUILD_TREE
```

After building the binary tree, the algorithm determines the position of each element in the sorted array. It traverses the tree and keeps a count of the number of elements in the left and right subtrees of any element. Finally, each element is placed in its proper position in time Q(1), and the array is sorted. The algorithm that traverses the binary tree and computes the position of each element is left as an exercise (Problem 9.15). The average run time of this algorithm is Q(log n) on an n-process PRAM. Thus, its overall process-time product is Q(n log n), which is cost-optimal.

Parallel Formulation for Practical Architectures

We now turn our attention to a more realistic parallel architecture – that of a p-process system connected via an interconnection network. Initially, our discussion will focus on developing an algorithm for a shared-address-space system and then we will show how this algorithm can be adapted to message-passing systems.

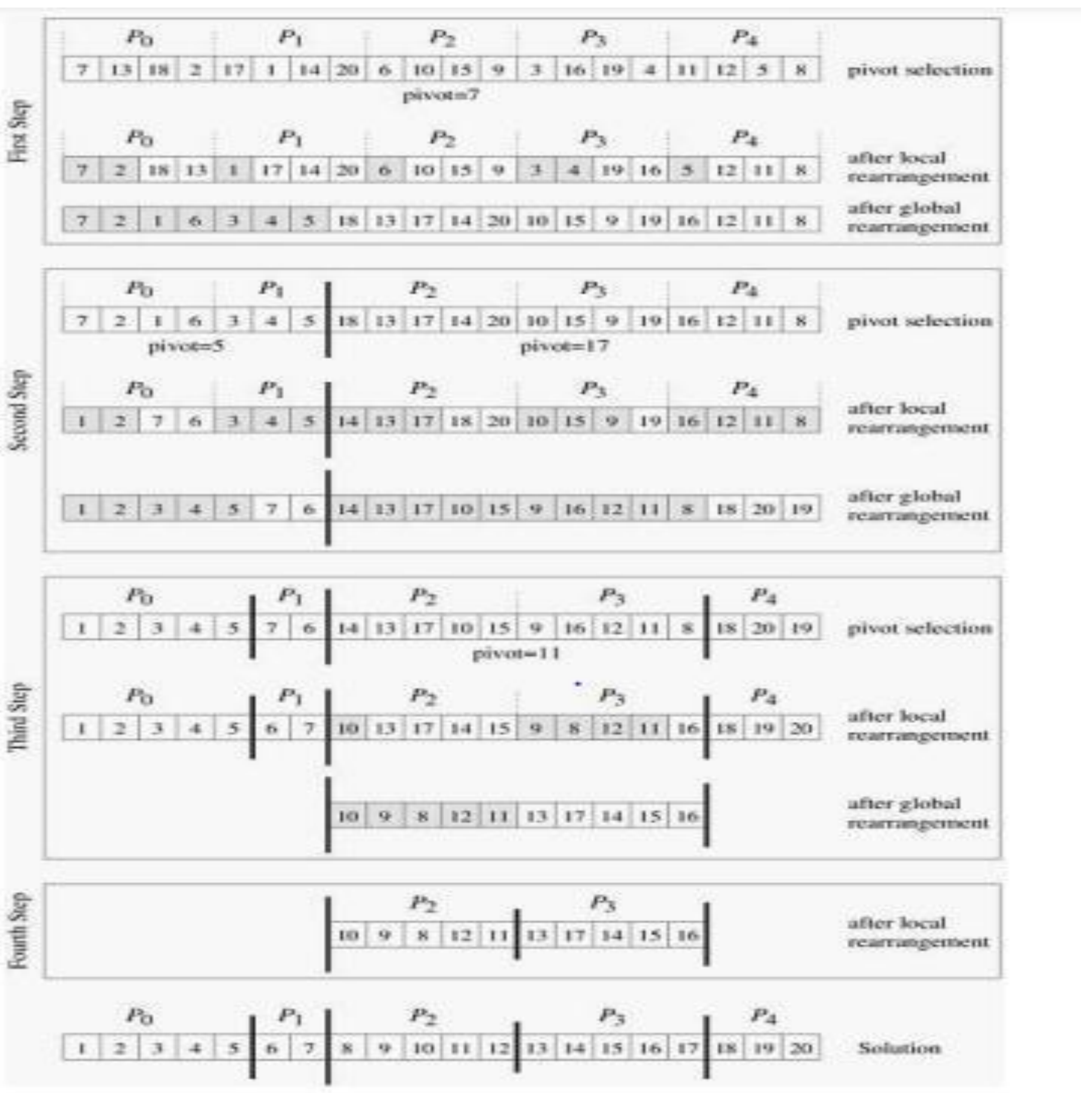
Shared-Address-Space Parallel Formulation

The quicksort formulation for a shared-address-space system works as follows. Let A be an array of n elements that need to be sorted and p be the number of processes. Each process is assigned a consecutive block of n/p elements, and the labels of the processes define the global order of the sorted sequence. Let Ai be the block of elements assigned to process Pi . The algorithm starts by selecting a pivot element, which is broadcast to all processes. Each process Pi, upon receiving the pivot, rearranges its assigned block of elements into two sub blocks, one with elements smaller than the pivot Si and one with elements larger than the pivot Li. This local rearrangement is done in place using the collapsing the loops approach of quicksort. The next step of the algorithm is to rearrange the elements of the original array A so that all the elements that are smaller than the pivot (i.e.,) are stored at the beginning of the array, and all the elements that are larger than the pivot (i.e.,) are stored at the end of the array. Once this global rearrangement is done, then the algorithm proceeds to partition the processes into two groups, and assign to the first group the task of sorting the smaller elements S, and to the second group the task of sorting the larger elements L . Each of these steps is performed by recursively calling the parallel quicksort algorithm. Note that by simultaneously partitioning both the processes and the original array each group of processes can proceed independently. The recursion ends when a particular sub-block of elements is assigned to only a single process, in which case the process sorts the elements using a serial quicksort algorithm. The partitioning of processes into two groups is done according to the relative sizes of the S and L blocks. In particular, the first processes are assigned to sort the smaller elements S, and the rest of the processes are assigned to sort the larger elements L. Note that the 0.5 term in the above formula is to ensure that the processes are assigned in the most balanced fashion.

Example 9.1 Efficient parallel quicksort

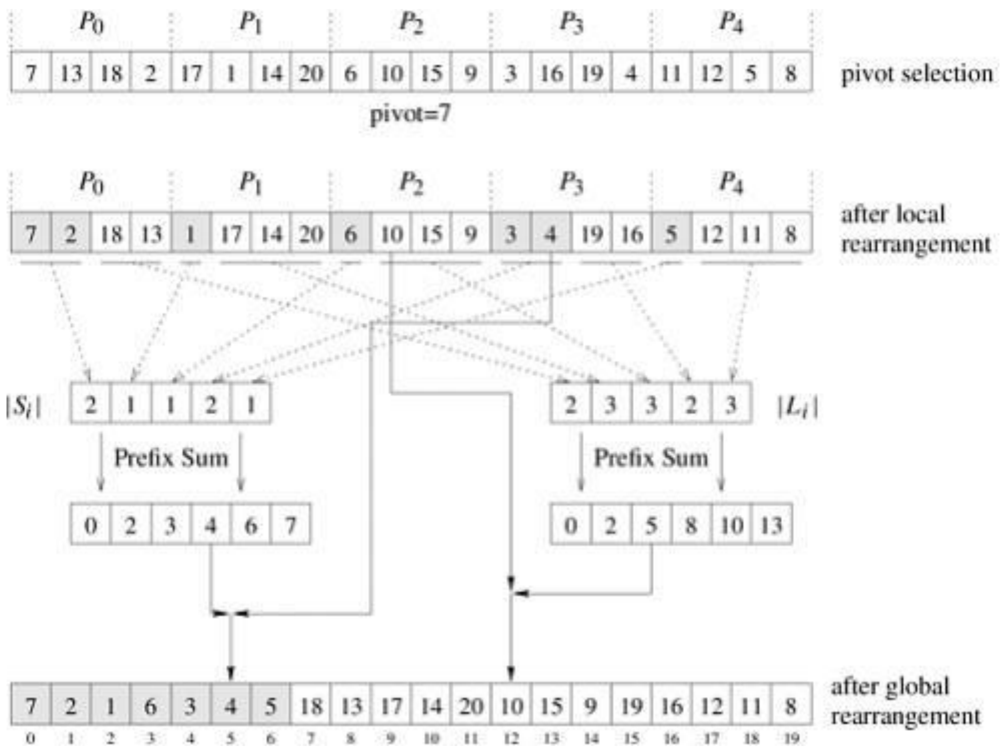
Figure 9.18 illustrates this algorithm using an example of 20 integers and five processes. In the first step, each process locally rearranges the four elements that it is initially responsible for, around the pivot element (seven in this example), so that the elements smaller or equal to the pivot are moved to the beginning of the locally assigned portion of the array (and are shaded in the figure). Once this local rearrangement is done, the processes perform a global rearrangement to obtain the third array shown in the figure (how this is performed will be discussed shortly). In the second step, the processes are partitioned into two groups. The first contains {P0, P1} and is responsible for sorting the elements that are smaller than or equal to seven, and the second group contains processes {P2, P3, P4} and is responsible for sorting the elements that are greater than seven. Note that the sizes of these process groups were created to match the relative size of the smaller than and larger than the pivot arrays. Now, the steps of pivot selection, local, and global rearrangement are recursively repeated for each process group and sub-array, until a sub-array is assigned to a single process, in which case it proceeds to sort it locally. Also note that these final local sub-arrays will in general be of different size, as they depend on the elements that were selected to act as pivots

Figure 9.18. An example of the execution of an efficient shared address-space quicksort algorithm.



In order to globally rearrange the elements of A into the smaller and larger sub-arrays we need to know where each element of A will end up going at the end of that rearrangement. One way
of doing this rearrangement is illustrated at the bottom of Figure 9.19. In this approach, S is obtained by concatenating the various Si blocks over all the processes, in increasing order of process label. Similarly, L is obtained by concatenating the various Li blocks in the same order. As a result, for process Pi, the j th element of its Si sub-block will be stored at location $\sum_{k=0}^{i-1} |S_k| + j$, and the j th element of its Li sub-block will be stored at location $\sum_{k=i}^{p-1} |L_k| - j$.

Figure 9.19. Efficient global rearrangement of the array.



These locations can be easily computed using the prefix-sum operation described in Section 4.3. Two prefix-sums are computed, one involving the sizes of the Si sub-blocks and the other the sizes of the Li sub-blocks. Let Q and R be the arrays of size p that store these prefix sums, respectively. Their elements will be

$$Q_i = \sum_{k=0}^{i-1} S_i, \quad \text{and} \quad R_i = \sum_{k=0}^{i-1} L_i.$$

Note that for each process Pi, Qi is the starting location in the final array where its lower-thanthe-pivot element will be stored, and Ri is the ending location in the final array where its greater-than-the-pivot elements will be stored. Once these locations have been determined, the overall rearrangement of A can be easily performed by using an auxiliary array A' of size n. These steps are illustrated in Figure 9.19. Note that the above definition of prefix-sum is slightly different from that described in Section 4.3, in the sense that the value that is computed for location Qi (or Ri) does not include Si (or Li) itself. This type of prefix-sum is sometimes referred to as non-inclusive prefix-sum.

Analysis The complexity of the shared-address-space formulation of the quicksort algorithm depends on two things. The first is the amount of time it requires to split a particular array into the smaller-than- and the greater-than-the-pivot sub-arrays, and the second is the degree to which the various pivots being selected lead to balanced partitions. In this section, to simplify our analysis, we will assume that pivot selection always results in balanced partitions. However the issue of proper pivot selection and its impact on the overall parallel performance is addressed in Section 9.4.4.

Given an array of n elements and p processes, the shared-address-space formulation of the quicksort algorithm needs to perform four steps: (i) determine and broadcast the pivot; (ii) locally rearrange the array assigned to each process; (iii) determine the locations in the globally rearranged array that the local elements will go to; and (iv) perform the global rearrangement. The first step can be performed in time Q(log p) using an efficient recursive doubling approach for shared-address-space broadcast. The second step can be done in time Q(n/p) using the traditional quicksort algorithm for splitting around a pivot element. The third step can be done in Q(log p) using two prefix sum operations. Finally, the fourth step can be done in at least time Q(n/p) as it requires us to copy the local elements to their final destination. Thus, the overall complexity of splitting an n-element array is Q(n/p) + Q(log p). This process is repeated for each of the two subarrays recursively on half the processes, until the array is split into p parts, at which point each process sorts the elements of the array assigned to it using the serial quicksort algorithm. Thus, the overall complexity of the parallel algorithm is:

Equation 9.8

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right) + \Theta(\log^2 p)}^{\text{array splits}}.$$

The communication overhead in the above formulation is reflected in the Q(log2 p) term, which leads to an overall isoefficiency of Q(p log2 p). It is interesting to note that the overall scalability of the algorithm is determined by the amount of time required to perform the pivot broadcast and the prefix sum operations.

Message-Passing Parallel Formulation

The quicksort formulation for message-passing systems follows the general structure of the shared-address-space formulation. However, unlike the shared-address-space case in which array A and the globally rearranged array A' are stored in shared memory and can be accessed by all the processes, these arrays are now explicitly distributed among the processes. This makes the task of splitting A somewhat more involved. In particular, in the message-passing version of the parallel quicksort, each process stores n/p elements of array A. This array is also partitioned around a particular pivot element using a two-phase approach. In the first phase (which is similar to the shared-address-space formulation), the locally stored array Ai at process Pi is partitioned into the smaller-than- and larger-than-the-pivot sub-arrays Si and Li locally. In the next phase, the algorithm first determines which processes will be responsible for recursively sorting the smaller-than-thepivot sub-arrays (i.e.,) and which process will be responsible for recursively sorting the larger-than-the-pivot sub-arrays (i.e.,). Once this is done, then the processes send their Si and Li arrays to the corresponding processes. After that, the processes are partitioned into the two groups, one for S and one for L, and the algorithm proceeds recursively. The recursion terminates when each sub-array is assigned to a single process, at which point it is sorted locally.

The method used to determine which processes will be responsible for sorting S and L is identical to that for the shared-address-space formulation, which tries to partition the processes to match the relative size of the two sub-arrays. Let pS and pL be the number of processesassigned to sort S and L, respectively. Each one of the pS processes will end up storing |S|/pS elements of the smaller-than-the-pivot sub-array, and each one of the pL processes will end up storing |L|/pL elements of the larger-than-the-pivot sub-array. The method used to determine where each process Pi will send its Si and Li elements follows the same overall strategy as the shared-address-space formulation. That is, the various Si (or Li) sub-arrays will be stored in consecutive locations in S (or L) based on the process number. The actual processes that will be responsible for these elements are determined by the partition of S (or L) into pS (or pL) equalsize segments, and can be computed using a prefix-sum operation. Note that each process Pi may need to split its Si (or Li) sub-arrays into multiple segments and send each one to different processes. This can happen because its elements may be assigned to locations in S (or L) that span more than one process. In general, each process may have to send its elements to two different processes; however, there may be cases in which more than two partitions are required.

Analysis Our analysis of the message-passing formulation of quicksort will mirror the corresponding analysis of the shared-address-space formulation.

Consider a message-passing parallel computer with p processes and O (p) bisection bandwidth. The amount of time required to split an array of size n is Q(log p) for broadcasting the pivot element, Q(n/p) for splitting the locally assigned portion of the array, Q(log p) for performing the prefix sums to determine the process partition sizes and the destinations of the various Si and Li sub-arrays, and the amount of time required for sending and receiving the various arrays. This last step depends on how the processes are mapped on the underlying architecture and on the maximum number of processes that each process needs to communicate with. In general, this communication step involves all-to-all personalized communication (because a particular process may end-up receiving elements from all other processes), whose complexity has a lower bound of Q(n/p). Thus, the overall complexity for the split is Q(n/p) + Q(log p), which is asymptotically similar to that of the shared-address-space formulation. As a result, the overall runtime is also the same as in Equation 9.8, and the algorithm has a similar isoefficiency function of Q(p log2 p).

Pivot Selection

In the parallel quicksort algorithm, we glossed over pivot selection. Pivot selection is particularly difficult, and it significantly affects the algorithm's performance. Consider the case in which the first pivot happens to be the largest element in the sequence. In this case, after the first split, one of the processes will be assigned only one element, and the remaining p - 1 processes will be assigned n - 1 elements. Hence, we are faced with a problem whose size has been reduced only by one element but only p - 1 processes will participate in the sorting operation. Although this is a contrived example, it illustrates a significant problem with parallelizing the quicksort algorithm. Ideally, the split should be done such that each partition has a non-trivial fraction of the original array.

One way to select pivots is to choose them at random as follows. During the i th split, one process in each of the process groups randomly selects one of its elements to be the pivot for this partition. This is analogous to the random pivot selection in the sequential quicksort algorithm. Although this method seems to work for sequential quicksort, it is not well suited to the parallel formulation. To see this, consider the case in which a bad pivot is selected at some point. In sequential quicksort, this leads to a partitioning in which one subsequence is significantly larger than the other. If all subsequent pivot selections are good, one poor pivot will increase the overall work by at most an amount equal to the length of the subsequence; thus, it will not significantly degrade the performance of sequential quicksort. In the parallel formulation, however, one poor pivot may lead to a partitioning in which a process becomes idle, and that will persist throughout the execution of the algorithm.

If the initial distribution of elements in each process is uniform, then a better pivot selection method can be derived. In this case, the n/p elements initially stored at each process form a representative sample of all n elements. In other words, the median of each n/p-element subsequence is very close to the median of the entire n-element sequence. Why is this a good pivot selection scheme under the assumption of identical initial distributions? Since the distribution of elements on each process is the same as the overall distribution of the n elements, the median selected to be the pivot during the first step is a good approximation of the overall median. Since the selected pivot is very close to the overall median, roughly half of the elements in each process are smaller and the other half larger than the pivot. Therefore, the first split leads to two partitions, such that each of them has roughly n/2 elements. Similarly, the elements assigned to each process of the group that is responsible for sorting the smallerthan-the-pivot elements (and the group responsible for sorting the larger-than-the-pivot elements) have the same distribution as the n/2 smaller (or larger) elements of the original list. Thus, the split not only maintains load balance but also preserves the assumption of uniform element distribution in the process group. Therefore, the application of the same pivot selection scheme to the sub-groups of processes continues to yield good pivot selection.

Can we really assume that the n/p elements in each process have the same distribution as the overall sequence? The answer depends on the application. In some applications, either the random or the median pivot selection scheme works well, but in others neither scheme delivers good performance.

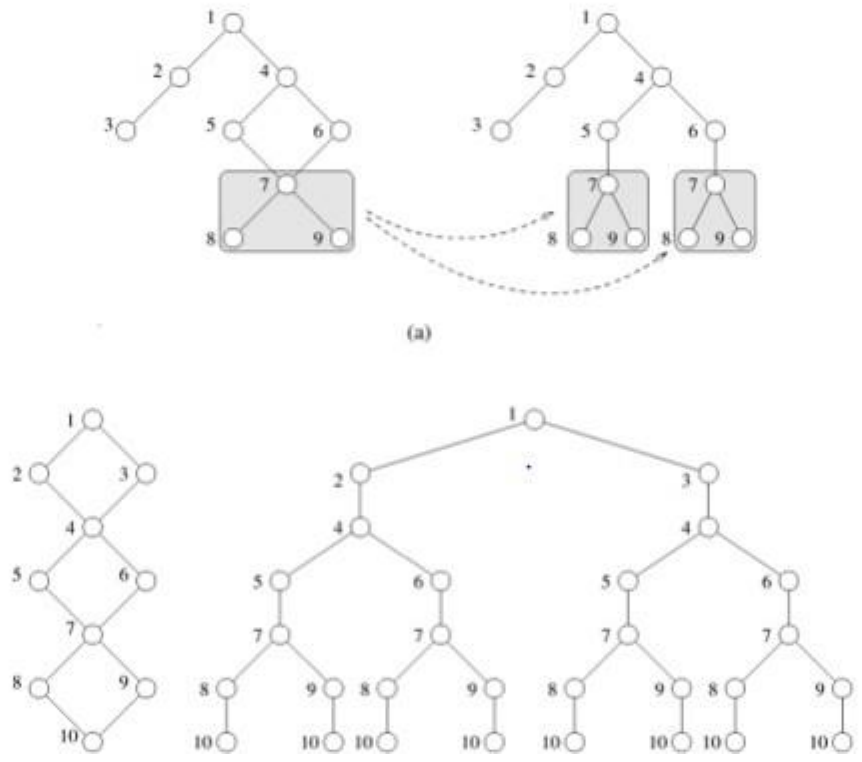
UNIT-V

Search Algorithms for Discrete Optimization Problems: Sequential Search Algorithms, Parallel Depth-First Search, Parallel Best-First Search, Speed up Anomalies in Parallel Search Algorithms

Sequential Search Algorithms:

The most suitable sequential search algorithm to apply to a state space depends on whether the space forms a graph or a tree. In a tree, each new successor leads to an unexplored part of the search space. An example of this is the 0/1 integer-programming problem. In a graph, however, a state can be reached along multiple paths. An example of such a problem is the 8- puzzle. For such problems, whenever a state is generated, it is necessary to check if the state has already been generated. If this check is not performed, then effectively the search graph is unfolded into a tree in which a state is repeated for every path that leads to it (Figure 11.3).

Figure 11.3. Two examples of unfolding a graph into a tree.



For many problems (for example, the 8-puzzle), unfolding increases the size of the search space by a small factor. For some problems, however, unfolded graphs are much larger than the original graphs. Figure 11.3(b) illustrates a graph whose corresponding tree has an exponentially higher number of states. In this section, we present an overview of various sequential algorithms used to solve DOPs that are formulated as tree or graph search problems.

Depth-First Search Algorithms:

Depth-first search (DFS) algorithms solve DOPs that can be formulated as tree-search problems. DFS begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), then DFS backtracks and expands a different node. In some DFS algorithms, successors of a node are expanded in an order determined by their heuristic values. A major advantage of DFS is that its storage requirement is linear in the depth of the state space being searched. The following sections discuss three algorithms based on depth-first search.

Simple Backtracking

Simple backtracking is a depth-first search method that terminates upon finding the first solution. Thus, it is not guaranteed to find a minimum-cost solution. Simple backtracking uses no heuristic information to order the successors of an expanded node. A variant, ordered backtracking, does use heuristics to order the successors of an expanded node.

Depth-First Branch-and-Bound

Depth-first branch-and-bound (DFBB) exhaustively searches the state space; that is, it continues to search even after finding a solution path. Whenever it finds a new solution path, it updates the current best solution path. DFBB discards inferior partial solution paths (that is, partial solution paths whose extensions are guaranteed to be worse than the current best solution path). Upon termination, the current best solution is a globally optimal solution.

Iterative Deepening A*

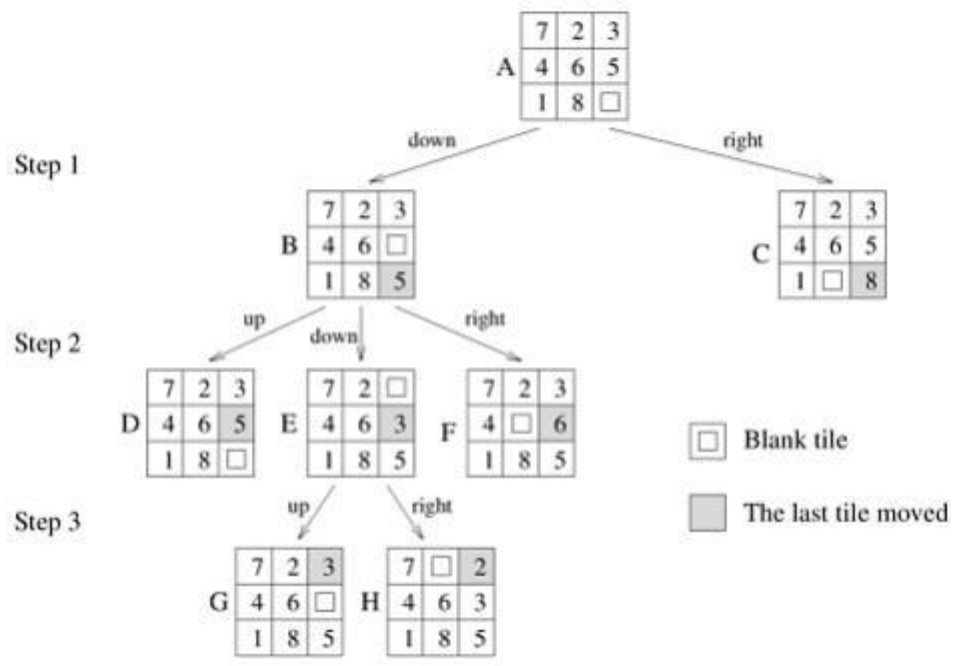
Trees corresponding to DOPs can be very deep. Thus, a DFS algorithm may get stuck searching a deep part of the search space when a solution exists higher up on another branch. For such trees, we impose a bound on the depth to which the DFS algorithm searches. If the node to be expanded is beyond the depth bound, then the node is not expanded and the algorithm backtracks. If a solution is not found, then the entire state space is searched again using a larger depth bound. This technique is called iterative deepening depth-first search (IDDFS). Note that this method is guaranteed to find a solution path with the fewest edges. However, it is not guaranteed to find a least-cost path.

Iterative deepening A* (IDA*) is a variant of ID-DFS. IDA* uses the l-values of nodes to bound depth (recall from Section 11.1 that for node x, $l(x) = g(x) + h(x)$). IDA* repeatedly performs cost-bounded DFS over the search space. In each iteration, IDA* expands nodes depth-first. If the l-value of the node to be expanded is greater than the cost bound, then IDA* backtracks. If a solution is not found within the current cost bound, then IDA* repeats the entire depth-first search using a higher cost bound. In the first iteration, the cost bound is set to the lvalue of the initial state s. Note that since g(s) is zero, l(s) is equal to h(s). In each subsequent iteration, the cost bound is increased. The new cost bound is equal to the minimum l-value of the nodes that were generated but could not be expanded in the previous iteration. The algorithm terminates when a goal node is expanded. IDA* is guaranteed to find an optimal solution if the heuristic function is admissible. It may appear that IDA* performs a lot of redundant work across iterations. However, for many problems the redundant work performed by IDA* is minimal, because most of the work is done deep in the search space.

Example 11.5 Depth-first search: the 8-puzzle

Figure 11.4 shows the execution of depth-first search for solving the 8-puzzle problem. The search starts at the initial configuration. Successors of this state are generated by applying possible moves. During each step of the search algorithm a new state is selected, and its successors are generated. The DFS algorithm expands the deepest node in the tree. In step 1, the initial state A generates states B and C. One of these is selected according to a predetermined criterion. In the example, we order successors by applicable moves as follows: up, down, left, and right. In step 2, the DFS algorithm selects state B and generates states D, E, and F. Note that the state D can be discarded, as it is a duplicate of the parent of B. In step 3, state E is expanded to generate states G and H. Again G can be discarded because it is a duplicate of B. The search proceeds in this way until the algorithm backtracks or the final configuration is generated.

Figure 11.4. States resulting from the first three steps of depth-first search applied to an instance of the 8-puzzle.

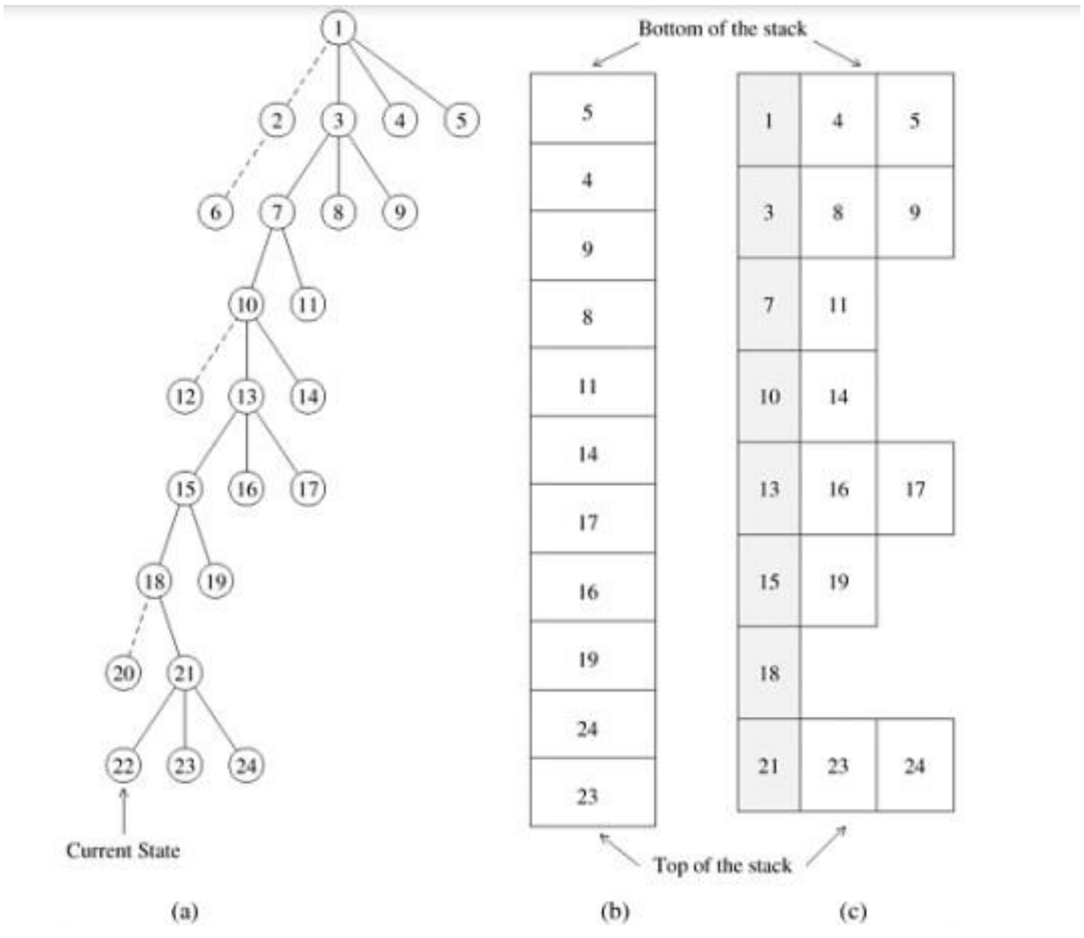


In each step of the DFS algorithm, untried alternatives must be stored. For example, in the 8- puzzle problem, up to three untried alternatives are stored at each step. In general, if m is the amount of storage required to store a state, and d is the maximum depth, then the total space requirement of the DFS algorithm is O (md).

The state-space tree searched by parallel DFS can be efficiently represented as a stack. Since the depth of the stack increases linearly with the depth of the tree, the memory requirements of a stack representation are low.

There are two ways of storing untried alternatives using a stack. In the first representation, untried alternates are pushed on the stack at each step. The ancestors of a state are not represented on the stack. Figure 11.5(b) illustrates this representation for the tree shown in Figure 11.5(a). In the second representation, shown in Figure 11.5(c), untried alternatives are stored along with their parent state. It is necessary to use the second representation if the sequence of transformations from the initial state to the goal state is required as a part of the solution. Furthermore, if the state space is a graph in which it is possible to generate an ancestor state by applying a sequence of transformations to the current state, then it is desirable to use the second representation, because it allows us to check for duplication of ancestor states and thus remove any cycles from the state-space graph. The second representation is useful for problems such as the 8-puzzle. In Example 11.5, using the second representation allows the algorithm to detect that nodes D and G should be discarded.

Figure 11.5. Representing a DFS tree: (a) the DFS tree; successor nodes shown with dashed lines have already been explored; (b) the stack storing untried alternatives only; and (c) the stack storing untried alternatives along with their parent. The shaded blocks represent the parent state and the block to the right represents successor states that have not been explored.



11.2.2 Best-First Search Algorithms

Best-first search (BFS) algorithms can search both graphs and trees. These algorithms use heuristics to direct the search to portions of the search space likely to yield solutions. Smaller heuristic values are assigned to more promising nodes. BFS maintains two lists: open and closed. At the beginning, the initial node is placed on the open list. This list is sorted according to a heuristic evaluation function that measures how likely each node is to yield a solution. In each step of the search, the most promising node from the open list is removed. If this node is a goal node, then the algorithm terminates. Otherwise, the node is expanded. The expanded node is placed on the closed list. The successors of the newly expanded node are placed on the open list under one of the following circumstances: (1) the successor is not already on the open or closed lists, and (2) the successor is already on the open or closed list but has a lower heuristic value. In the second case, the node with the higher heuristic value is deleted.

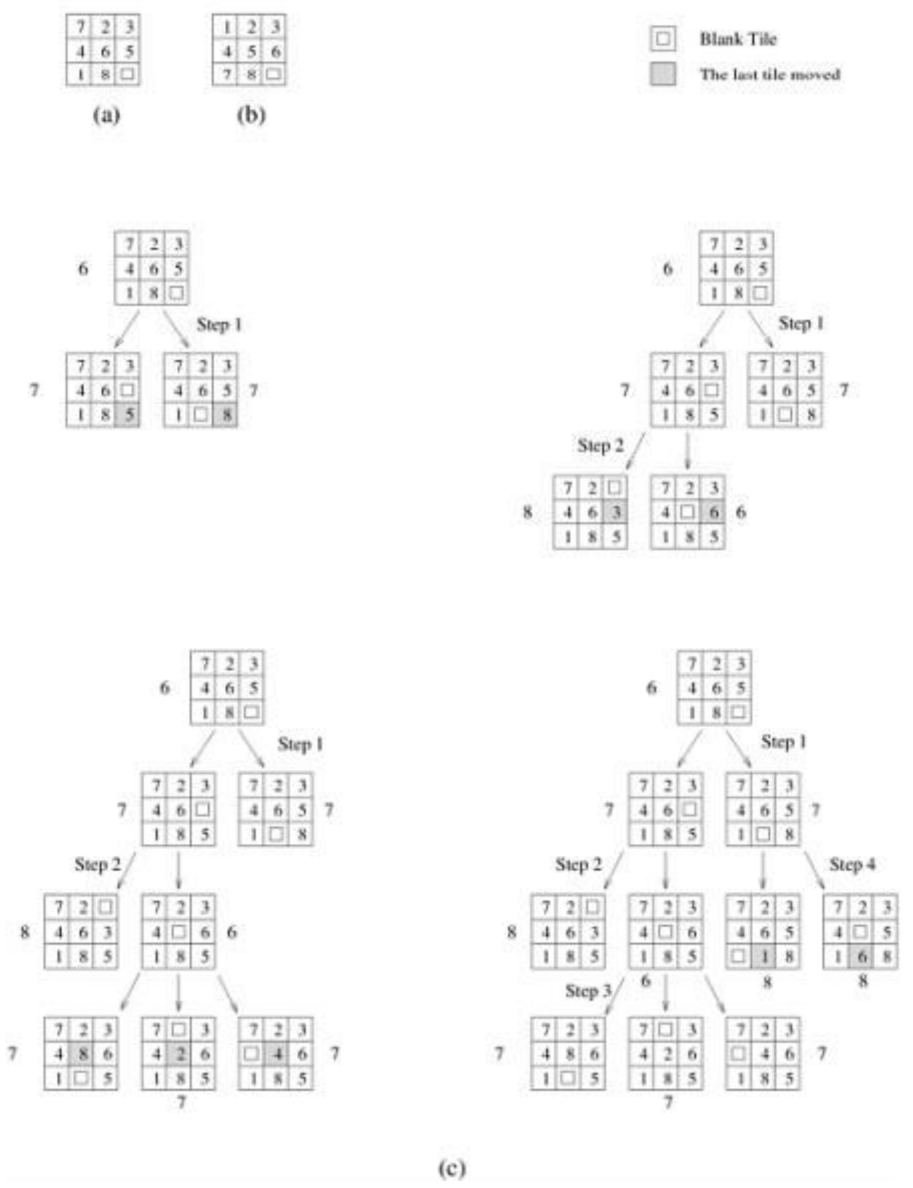
A common BFS technique is the A* algorithm. The A* algorithm uses the lower bound function l as a heuristic evaluation function. Recall from Section 11.1 that for each node x , l(x) is the sum of g(x) and h(x). Nodes in the open list are ordered according to the value of the l function. At each step, the node with the smallest l-value (that is, the best node) is removed from the open list and expanded. Its successors are inserted into the open list at the proper positions and the node itself is inserted into the closed list. For an admissible heuristic function, A* finds an optimal solution.

The main drawback of any BFS algorithm is that its memory requirement is linear in the size of the search space explored. For many problems, the size of the search space is exponential in the depth of the tree expanded. For problems with large search spaces, memory becomes a limitation.

Example 11.6 Best-first search: the 8-puzzle

Consider the 8-puzzle problem from Examples 11.2 and 11.4. Figure 11.6 illustrates four steps of best-first search on the 8-puzzle. At each step, a state x with the minimum l-value ($l(x) = g(x) + h(x)$) is selected for expansion. Ties are broken arbitrarily. BFS can check for a duplicate nodes, since all previously generated nodes are kept on either the open or closed list.

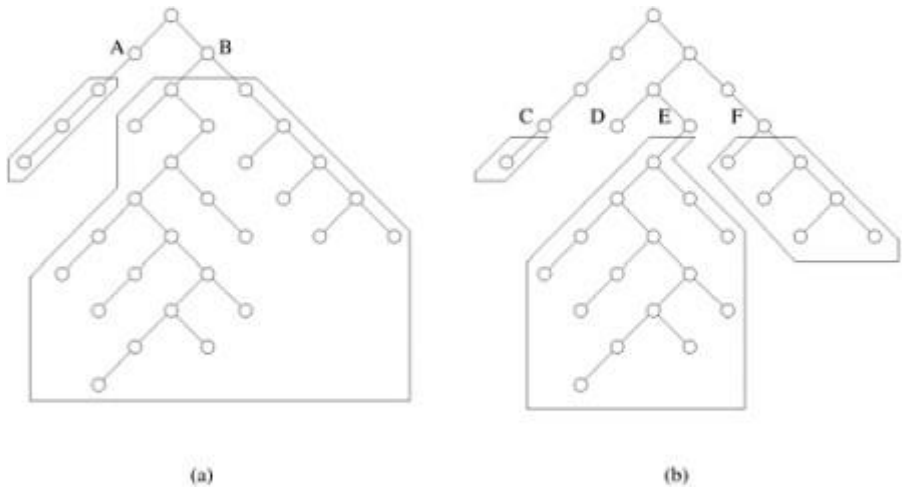
Figure 11.6. Applying best-first search to the 8-puzzle: (a) initial configuration; (b) final configuration; and (c) states resulting from the first four steps of best-first search. Each state is labeled with its h-value (that is, the Manhattan distance from the state to the final state).



Parallel Depth-First Search:

We start our discussion of parallel depth-first search by focusing on simple backtracking. Parallel formulations of depth-first branch-and-bound and IDA* are similar to those discussed in this section and are addressed in Sections 11.4.6 and 11.4.7. The critical issue in parallel depth-first search algorithms is the distribution of the search space among the processors. Consider the tree shown in Figure 11.7. Note that the left subtree (rooted at node A) can be searched in parallel with the right subtree (rooted at node B). By statically assigning a node in the tree to a processor, it is possible to expand the whole subtree rooted at that node without communicating with another processor. Thus, it seems that such a static allocation yields a good parallel search algorithm.

Figure 11.7. The unstructured nature of tree search and the imbalance resulting from static partitioning.



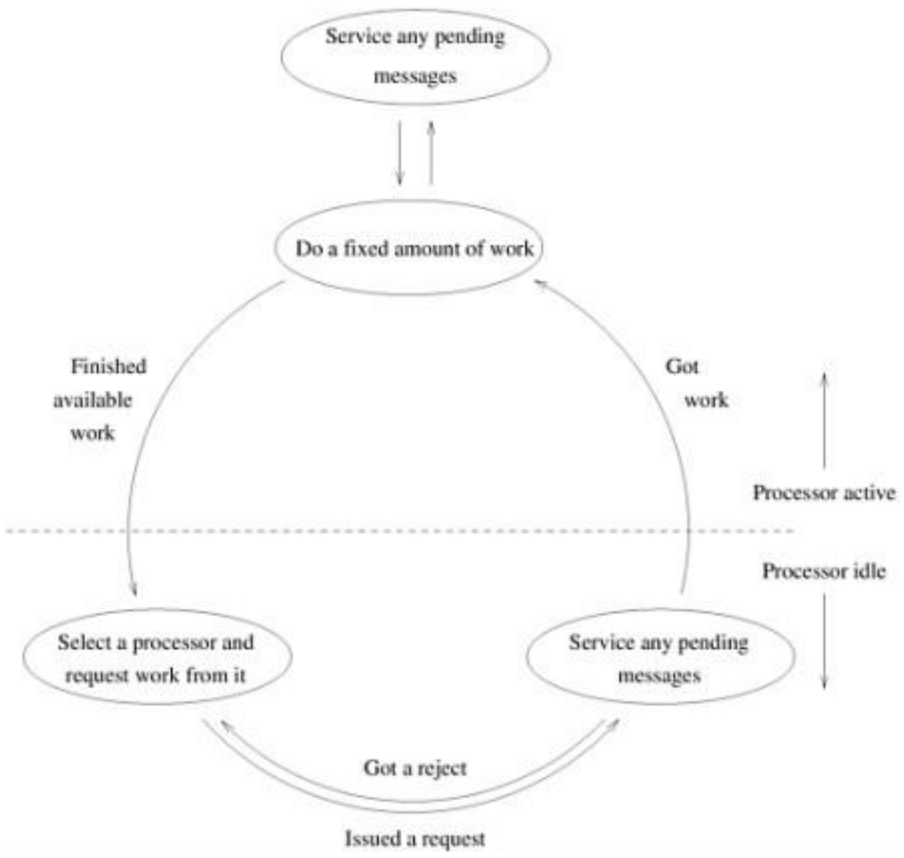
Let us see what happens if we try to apply this approach to the tree in Figure 11.7. Assume that we have two processors. The root node is expanded to generate two nodes (A and B), and each of these nodes is assigned to one of the processors. Each processor now searches the subtrees rooted at its assigned node independently. At this point, the problem with static node assignment becomes apparent. The processor exploring the subtree rooted at node A expands considerably fewer nodes than does the other processor. Due to this imbalance in the workload, one processor is idle for a significant amount of time, reducing efficiency. Using a larger number of processors worsens the imbalance. Consider the partitioning of the tree for four processors. Nodes A and B are expanded to generate nodes C, D, E, and F. Assume that each of these nodes is assigned to one of the four processors. Now the processor searching the subtree rooted at node E does most of the work, and those searching the subtrees rooted at nodes C and D spend most of their time idle. The static partitioning of unstructured trees yields poor performance because of substantial variation in the size of partitions of the search space rooted at different nodes. Furthermore, since the search space is usually generated dynamically, it is difficult to get a good estimate of the size of the search space beforehand. Therefore, it is necessary to balance the search space among processors dynamically.

In dynamic load balancing, when a processor runs out of work, it gets more work from another processor that has work. Consider the two-processor partitioning of the tree in Figure 11.7(a). Assume that nodes A and B are assigned to the two processors as we just described. In this case when the processor searching the subtree rooted at node A runs out of work, it requests work from the other processor. Although the dynamic distribution of work results in communication overhead for work requests and work transfers, it reduces load imbalance among processors. This section explores several schemes for dynamically balancing the load between processors.

A parallel formulation of DFS based on dynamic load balancing is as follows. Each processor performs DFS on a disjoint part of the search space. When a processor finishes searching its part of the search space, it requests an unsearched part from other processors. This takes the form of work request and response messages in message passing architectures, and locking and extracting work in shared address space machines. Whenever a processor finds a goal node, all the processors terminate. If the search space is finite and the problem has no solutions, then all the processors eventually run out of work, and the algorithm terminates.

Since each processor searches the state space depth-first, unexplored states can be conveniently stored as a stack. Each processor maintains its own local stack on which it executes DFS. When a processor's local stack is empty, it requests (either via explicit messages or by locking) untried alternatives from another processor's stack. In the beginning, the entire search space is assigned to one processor, and other processors are assigned null search spaces (that is, empty stacks). The search space is distributed among the processors as they request work. We refer to the processor that sends work as the donor processor and to the processor that requests and receives work as the recipient processor.

Figure 11.8. A generic scheme for dynamic load balancing.


Service any pending messages
Do a fixed amount of work
Finished available work
Got work
Processor active
Processor idle
Select a processor and request work from it
Service any pending messages
Got a reject
Issued a request

On message passing architectures, in the active state, a processor does a fixed amount of work (expands a fixed number of nodes) and then checks for pending work requests. When a work request is received, the processor partitions its work into two parts and sends one part to the requesting processor. When a processor has exhausted its own search space, it becomes idle. This process continues until a solution is found or until the entire space has been searched. If a solution is found, a message is broadcast to all processors to stop searching. A termination detection algorithm is used to detect whether all processors have become idle without finding a solution.

Important Parameters of Parallel DFS

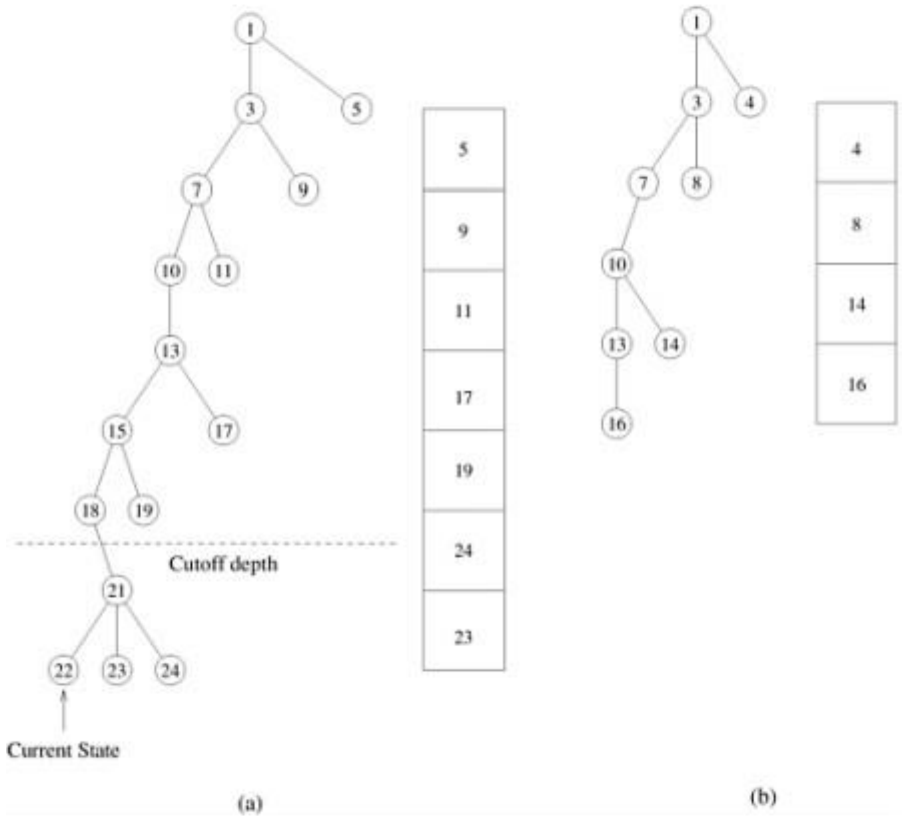
Two characteristics of parallel DFS are critical to determining its performance. First is the method for splitting work at a processor, and the second is the scheme to determine the donor processor when a processor becomes idle.

Work-Splitting Strategies

When work is transferred, the donor's stack is split into two stacks, one of which is sent to the recipient. In other words, some of the nodes (that is, alternatives) are removed from the donor's stack and added to the recipient's stack. If too little work is sent, the recipient quickly becomes idle; if too much, the donor becomes idle. Ideally, the stack is split into two equal pieces such that the size of the search space represented by each stack is the same. Such a split is called a half-split. It is difficult to get a good estimate of the size of the tree rooted at an unexpanded alternative in the stack. However, the alternatives near the bottom of the stack (that is, close to the initial node) tend to have bigger trees rooted at them, and alternatives near the top of the stack tend to have small trees rooted at them. To avoid sending very small amounts of work, nodes beyond a specified stack depth are not given away. This depth is called the cutoff depth some possible strategies for splitting the search space are (1) send nodes near the bottom of the stack, (2) send nodes near the cutoff depth, and (3) send half the nodes between the bottom of the stack and the cutoff depth. The suitability of a splitting strategy depends on the nature of the search space. If the search space is uniform, both strategies 1 and 3 work well. If the search space is highly irregular, strategy 3 usually works well. If a strong heuristic is available (to order successors so that goal nodes move to the left of the state-space tree), strategy 2 is likely to perform better, since it tries to distribute those parts of the search space likely to contain a solution. The cost of splitting also becomes important if the stacks are deep. For such stacks, strategy 1 has lower cost than strategies 2 and 3.

Figure 11.9 shows the partitioning of the DFS tree of Figure 11.5(a) into two subtrees using strategy 3. Note that the states beyond the cutoff depth are not partitioned. Figure 11.9 also shows the representation of the stack corresponding to the two subtrees.

Figure 11.9. Splitting the DFS tree in Figure 11.5. The two subtrees along with their stack representations are shown in (a) and (b).



Load-Balancing Schemes:

This section discusses three dynamic load-balancing schemes: asynchronous round robin, global round robin, and random polling. Each of these schemes can be coded for message passing as well as shared address space machines.

Asynchronous Round Robin In asynchronous round robin (ARR), each processor maintains an independent variable, target. Whenever a processor runs out of work, it uses target as the label of a donor processor and attempts to get work from it. The value of target is incremented (modulo p) each time a work request is sent. The initial value of target at each processor is set to ((label + 1) modulo p) where label is the local processor label. Note that work requests are generated independently by each processor. However, it is possible for two or more processors to request work from the same donor at nearly the same time.

Global Round Robin Global round robin (GRR) uses a single global variable called target. This variable can be stored in a globally accessible space in shared address space machines or at a designated processor in message passing machines. Whenever a processor needs work, it requests and receives the value of target, either by locking, reading, and unlocking on shared address space machines or by sending a message requesting the designated processor (say P0). The value of target is incremented (modulo p) before responding to the next request. The recipient processor then attempts to get work from a donor processor whose label is the value of target. GRR ensures that successive work requests are distributed evenly over all processors. A drawback of this scheme is the contention for access to target.

Random Polling Random polling (RP) is the simplest load-balancing scheme. When a processor becomes idle, it randomly selects a donor. Each processor is selected as a donor with equal probability, ensuring that work requests are evenly distributed.

A General Framework for Analysis of Parallel DFS

To analyze the performance and scalability of parallel DFS algorithms for any load-balancing scheme, we must compute the overhead To of the algorithm. Overhead in any load-balancing scheme is due to communication (requesting and sending work), idle time (waiting for work), termination detection, and contention for shared resources. If the search overhead factor is greater than one (i.e., if parallel search does more work than serial search), this will add another term to To. In this section we assume that the search overhead factor is one, i.e., the serial and parallel versions of the algorithm perform the same amount of computation.

For the load-balancing schemes discussed in Section 11.4.1, idle time is subsumed by communication overhead due to work requests and transfers. When a processor becomes idle, it immediately selects a donor processor and sends it a work request. The total time for which the processor remains idle is equal to the time for the request to reach the donor and for the reply to arrive. At that point, the idle processor either becomes busy or generates another work request. Therefore, the time spent in communication subsumes the time for which a processor is idle. Since communication overhead is the dominant overhead in parallel DFS, we now consider a method to compute the communication overhead for each load-balancing scheme.

It is difficult to derive a precise expression for the communication overhead of the loadbalancing schemes for DFS because they are dynamic. This section describes a technique that provides an upper bound on this overhead. We make the following assumptions in the analysis.

1. The work at any processor can be partitioned into independent pieces as long as its size exceeds a threshold .

2. A reasonable work-splitting mechanism is available. Assume that work w at one processor 1. is partitioned into two parts: yw and (1 - y)w for 0 y 1. Then there exists an arbitrarily small constant a (0 < a 0.5), such that yw > aw and (1 - y)w > aw. We call such a splitting mechanism a-splitting. The constant a sets a lower bound on the load imbalance that results from work splitting: both partitions of w have at least aw work.

The first assumption is satisfied by most depth-first search algorithms. The third work-splitting strategy described in Section 11.4.1 results in a-splitting even for highly irregular search spaces.

Analysis of Load-Balancing Schemes:

This section analyzes the performance of the load-balancing schemes introduced in Section 11.4.1. In each case, we assume that work is transferred in fixed-size messages (the effect of relaxing this assumption is explored in Problem 11.3). Recall that the cost of communicating an m-word message in the simplified cost model is tcomm = ts + twm. Since the message size m is assumed to be a constant, tcomm = O (1) if there is no congestion on the interconnection network. The communication overhead To (Equation 11.2) reduces to

Equation 11.3

$$T_o = O(V(p) \log W).$$

We balance this overhead with problem size *W* for each load-balancing scheme to derive the isoefficiency function due to communication.

Asynchronous Round Robin As discussed in Section 11.4.2, *V* (*p*) for ARR is *O* (p^2). Substituting into Equation 11.3, communication overhead T_o is given by *O* (p^2 log *W*). Balancing communication overhead against problem size *W*, we have

$$W = O(p^2 \log W).$$

Substituting *W* into the right-hand side of the same equation and simplifying,

$$\begin{aligned} W &= O(p^2 \log(p^2 \log W)), \\ &= O(p^2 \log p + p^2 \log \log W). \end{aligned}$$

The double-log term (log log W) is asymptotically smaller than the first term, provided p grows no slower than log W, and can be ignored. The isoefficiency function for this scheme is therefore given by O (p2 log p).

Equation 11.4

$$\frac{W}{p} = O(p \log W)$$

We can simplify Equation 11.4 to express W in terms of p. This yields an isoefficiency term of O (p2 log p).

Termination Detection

Dijkstra's Token Termination Detection Algorithm:

Consider a simplified scenario in which once a processor goes idle, it never receives more work. Visualize the p processors as being connected in a logical ring (note that a logical ring can be easily mapped to underlying physical topologies). Processor P0 initiates a token when it becomes idle. This token is sent to the next processor in the ring, P1. At any stage in the computation, if a processor receives a token, the token is held at the processor until the computation assigned to the processor is complete. On completion, the token is passed to the next processor in the ring. If the processor was already idle, the token is passed to the next processor. Note that if at any time the token is passed to processor Pi , then all processors P0, ..., Pi -1 have completed their computation. Processor Pp-1 passes its token to processor P0; when it receives the token, processor P0 knows that all processors have completed their computation and the algorithm can terminate. Such a simple scheme cannot be applied to the search algorithms described in this chapter, because after a processor goes idle, it may receive more work from other processors. The token termination detection scheme thus must be modified.

1. When it becomes idle, processor P0 initiates termination detection by making itself white and sending a white token to processor P1.
2. If processor Pj sends work to processor Pi and j > i then processor Pj becomes black.
3. If processor Pi has the token and Pi is idle, then it passes the token to Pi +1. If Pi is black, then the color of the token is set to black before it is sent to Pi +1. If Pi is white, the token is passed unchanged.
4. After Pi passes the token to Pi+1, Pi becomes white.

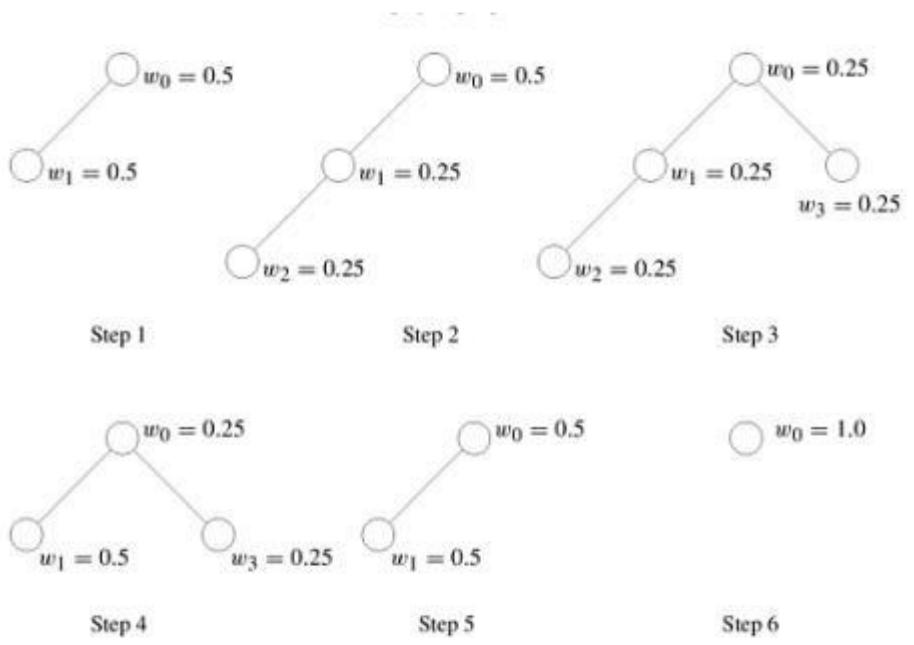
Tree-Based Termination Detection

Tree-based termination detection associates weights with individual work pieces. Initially processor P0 has all the work and a weight of one is associated with it. When its work is partitioned and sent to another processor, processor P0 retains half of the weight and gives half of it to the processor receiving the work. If Pi is the recipient processor and wi is the weight at processor Pi, then after the first work transfer, both w0 and wi are 0.5. Each time the work at a processor is partitioned, the weight is halved. When a processor completes its computation, it returns its weight to the processor from which it received work. Termination is signaled when the weight w0 at processor P0 becomes one and processor P0 has finished its work.

Example 11.7 Tree-based termination detection

Figure 11.10 illustrates tree-based termination detection for four processors. Initially, processor P0 has all the weight (w0 = 1), and the weight at the remaining processors is 0 (w1 = w2 = w3 = 0). In step 1, processor P0 partitions its work and gives part of it to processor P1. After this step, w0 and w1 are 0.5 and w2 and w3 are 0. In step 2, processor P1 gives half of its work to processor P2. The weights w1 and w2 after this work transfer are 0.25 and the weights w0 and w3 remain unchanged. In step 3, processor P3 gets work from processor P1 and the weights of all processors become 0.25. In step 4, processor P2 completes its work and sends its weight to processor P1. The weight w1 of processor P1 becomes 0.5. As processors complete their work, weights are propagated up the tree until the weight w0 at processor P0 becomes 1. At this point, all work has been completed and termination can be signaled.

Figure 11.10. Tree-based termination detection. Steps 1–6 illustrate the weights at various processors after each work transfer.



This termination detection algorithm has a significant drawback. Due to the finite precision of computers, recursive halving of the weight may make the weight so small that it becomes 0. In this case, weight will be lost and termination will never be signaled. This condition can be alleviated by using the inverse of the weights. If processor Pi has weight wi, instead of manipulating the weight itself, it manipulates 1/wi. The details of this algorithm are considered in Problem 11.5.

Parallel Best-First Search:

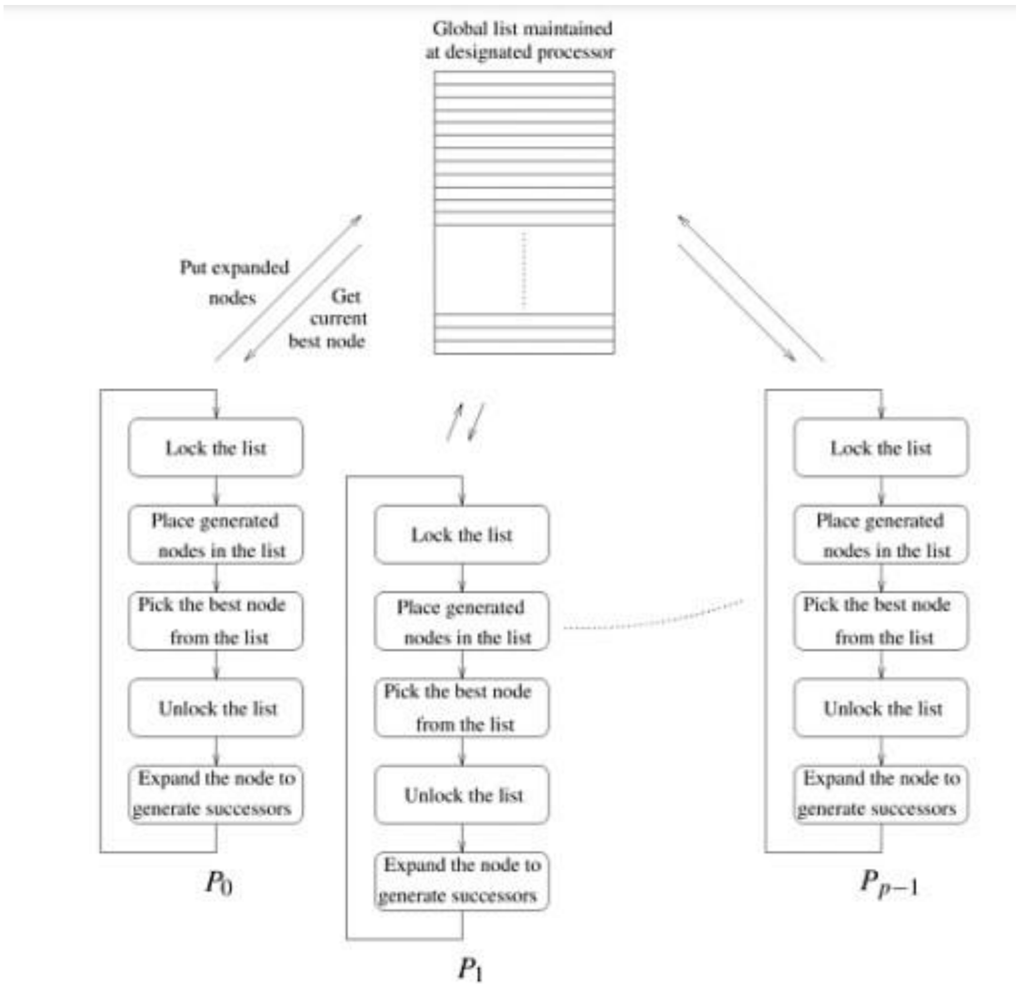
In most parallel formulations of BFS, different processors concurrently expand different nodes from the open list. These formulations differ according to the data structures they use to implement the open list. Given p processors, the simplest strategy assigns each processor to work on one of the current best nodes on the open list. This is called the centralized strategy because each processor gets work from a single global open list. Since this formulation of parallel BFS expands more than one node at a time, it may expand nodes that would not be expanded by a sequential algorithm. Consider the case in which the first node on the open list is a solution. The parallel formulation still expands the first p nodes on the open list. However, since it always picks the best p nodes, the amount of extra work is limited. Figure 11.14 illustrates this strategy. There are two problems with this approach:

1. The termination criterion of sequential BFS fails for parallel BFS. Since at any moment, p nodes from the open list are being expanded, it is possible that one of the nodes may be a solution that does not correspond to the best goal node (or the path found is not the shortest path). This is because the remaining p - 1 nodes may lead to search spaces containing better goal nodes. Therefore, if the cost of a solution found by a processor is c, then this solution is not guaranteed to correspond to the best goal node until

the cost of nodes being searched at other processors is known to be at least c. The termination criterion must be modified to ensure that termination occurs only after the best solution has been found.

2. Since the open list is accessed for each node expansion, it must be easily accessible to all processors, which can severely limit performance. Even on shared-address-space architectures, contention for the open list limits speedup. Let texp be the average time to expand a single node, and taccess be the average time to access the open list for a singlenode expansion. If there are n nodes to be expanded by both the sequential and parallel formulations (assuming that they do an equal amount of work), then the sequential run time is given by n(taccess + texp). Assume that it is impossible to parallelize the expansion of individual nodes. Then the parallel run time will be at least ntaccess, because the open list must be accessed at least once for each node expanded. Hence, an upper bound on the speedup is (taccess + texp)/taccess.

Figure 11.14. A general schematic for parallel best-first search using a centralized strategy. The locking operation is used here to serialize queue access by various processors.


Global list maintained at designated processor
Put expanded nodes
Get current best node
Lock the list
Place generated nodes in the list
Pick the best node from the list
Unlock the list
Expand the node to generate successors
P_0
P_1
P_{p-1}

One way to avoid the contention due to a centralized open list is to let each processor have a local open list. Initially, the search space is statically divided among the processors by expanding some nodes and distributing them to the local open lists of various processors. All the processors then select and expand nodes simultaneously. Consider a scenario where processors do not communicate with each other. In this case, some processors might explore parts of the search space that would not be explored by the sequential algorithm. This leads to a high search overhead factor and poor speedup. Consequently, the processors must communicate among themselves to minimize unnecessary search. The use of a distributed open list trades-off communication and computation: decreasing communication between distributed open lists increases search overhead factor, and decreasing search overhead factor with increased communication increases communication overhead.

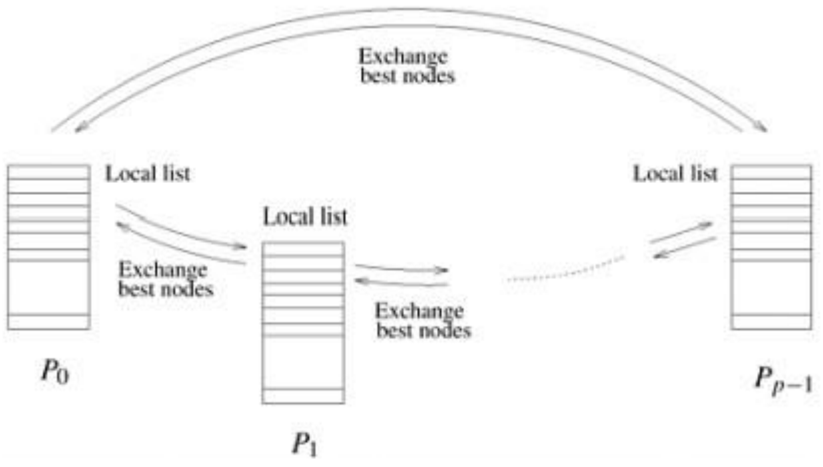
Communication Strategies for Parallel Best-First Tree Search

A communication strategy allows state-space nodes to be exchanged between open lists on different processors. The objective of a communication strategy is to ensure that nodes with good l-values are distributed evenly among processors. In this section we discuss three such strategies, as follows.

1. In the random communication strategy, each processor periodically sends some of its best nodes to the open list of a randomly selected processor. This strategy ensures that, if a processor stores a good part of the search space, the others get part of it. If nodes are transferred frequently, the search overhead factor can be made very small; otherwise it can become quite large. The communication cost determines the best node transfer frequency. If the communication cost is low, it is best to communicate after every node expansion.
2. In the ring communication strategy, the processors are mapped in a virtual ring. Each processor periodically exchanges some of its best nodes with the open lists of its neighbors in the ring. This strategy can be implemented on message passing as well as shared address space machines with the processors organized into a logical ring. As before, the cost of communication determines the node transfer frequency. Figure 11.15 illustrates the ring communication strategy.

Figure 11.15. A message-passing implementation of parallel bestfirst search using the ring communication strategy.

Figure 11.15. A message-passing implementation of parallel best-first search using the ring communication strategy.

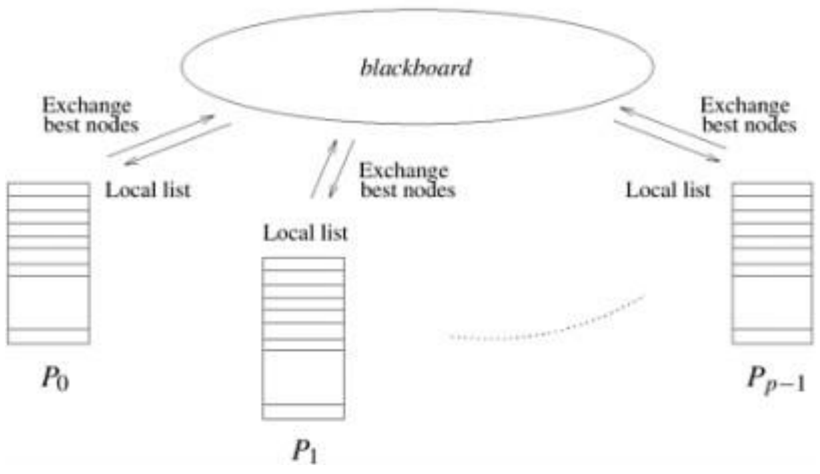


Unless the search space is highly uniform, the search overhead factor of this scheme is very high. The reason is that this scheme takes a long time to distribute good nodes from one processor to all other processors

3. In the blackboard communication strategy, there is a shared blackboard through which nodes are switched among processors as follows. After selecting the best node from its local open list, a processor expands the node only if its l-value is within a tolerable limit of the best node on the blackboard. If the selected node is much better than the best node on the blackboard, the processor sends some of its best nodes to the blackboard before expanding the current node. If the selected node is much worse than the best node on the blackboard, the processor retrieves some good nodes from the blackboard and reselects a node for expansion. Figure 11.16 illustrates the blackboard communication strategy. The blackboard strategy is suited only to shared-address-space computers, because the value of the best node in the blackboard has to be checked after each node expansion.

Figure 11.16. An implementation of parallel best-first search using 3. the blackboard communication strategy

the blackboard communication strategy.



Communication Strategies for Parallel Best-First Graph Search

While searching graphs, an algorithm must check for node replication. This task is distributed among processors. One way to check for replication is to map each node to a specific processor. Subsequently, whenever a node is generated, it is mapped to the same processor, which checks for replication locally. This technique can be implemented using a hash function that takes a node as input and returns a processor label. When a node is generated, it is sent to the processor whose label is returned by the hash function for that node. Upon receiving the node, a processor checks whether it already exists in the local open or closed lists. If not, the node is inserted in the open list. If the node already exists, and if the new node has a better cost associated with it, then the previous version of the node is replaced by the new node on the open list.

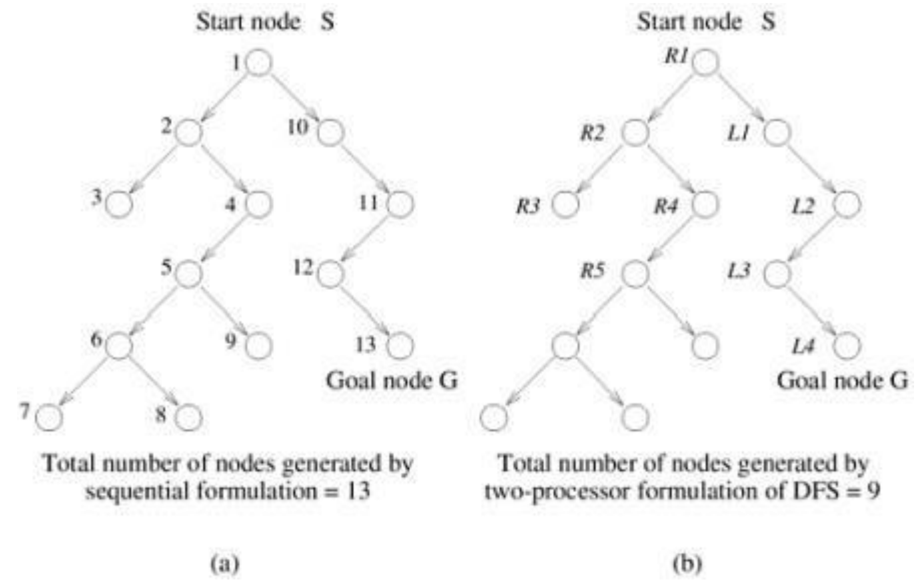
For a random hash function, the load-balancing property of this distribution strategy is similar to the random-distribution technique discussed in the previous section. This result follows from the fact that each processor is equally likely to be assigned a part of the search space that would also be explored by a sequential formulation. This method ensures an even distribution of nodes with good heuristic values among all the processors (Problem 11.10). However, hashing techniques degrade performance because each node generation results in communication.

Speedup Anomalies in Parallel Search Algorithms

In parallel search algorithms, speedup can vary greatly from one execution to another because the portions of the search space examined by various processors are determined dynamically and can differ for each execution. Consider the case of sequential and parallel DFS performed on the tree illustrated in Figure 11.17. Figure 11.17(a) illustrates sequential DFS search. The order of node expansions is indicated by node labels. The sequential formulation generates 13 nodes before reaching the goal node G.

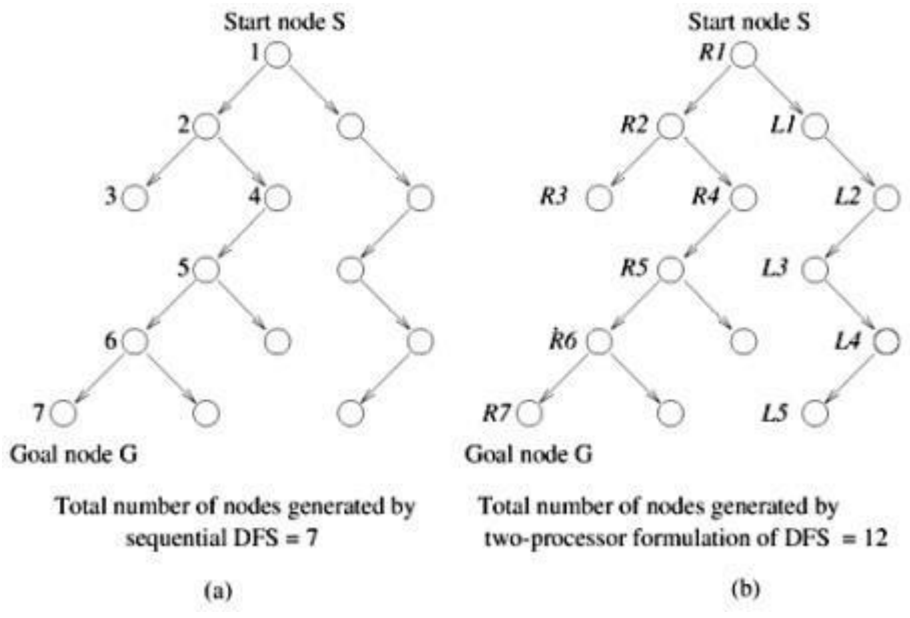
Figure 11.17. The difference in number of nodes searched by sequential and parallel formulations of DFS. For this example, parallel DFS reaches a goal node after searching fewer nodes than sequential

DFS.


Start node S
1
2
3
4
5
6
7
8
9
10
11
12
13
Goal node G
Total number of nodes generated by sequential formulation = 13
(a)
Start node S
R1
R2
R3
R4
R5
L1
L2
L3
L4
Goal node G
Total number of nodes generated by two-processor formulation of DFS = 9
(b)

Now consider the parallel formulation of DFS illustrated for the same tree in Figure 11.17(b) for two processors. The nodes expanded by the processors are labeled R and L. The parallel formulation reaches the goal node after generating only nine nodes. That is, the parallel formulation arrives at the goal node after searching fewer nodes than its sequential counterpart. In this case, the search overhead factor is 9/13 (less than one), and if communication overhead is not too large, the speedup will be superlinear.
Finally, consider the situation in Figure 11.18. The sequential formulation (Figure 11.18(a)) generates seven nodes before reaching the goal node, but the parallel formulation generates 12 nodes. In this case, the search overhead factor is greater than one, resulting in sublinear speedup.

Figure 11.18. A parallel DFS formulation that searches more nodes than its sequential counterpart.


Start node S
1
2
3
4
5
6
7
Goal node G
Total number of nodes generated by sequential DFS = 7
(a)
Start node S
R1
R2
R3
R4
R5
R6
R7
L1
L2
L3
L4
L5
Goal node G
Total number of nodes generated by two-processor formulation of DFS = 12
(b)

In summary, for some executions, the parallel version finds a solution after generating fewer nodes than the sequential version, making it possible to obtain superlinear speedup. For other executions, the parallel version finds a solution after generating more nodes, resulting in sublinear speedup. Executions yielding speedups greater than p by using p processors are referred to as acceleration anomalies. Speedups of less than p using p processors are called deceleration anomalies.

Analysis of Average Speedup in Parallel DFS:

In isolated executions of parallel search algorithms, the search overhead factor may be equal to one, less than one, or greater than one. It is interesting to know the average value of the search overhead factor. If it is less than one, this implies that the sequential search algorithm is not optimal. In this case, the parallel search algorithm running on a sequential processor (by emulating a parallel processor by using time-slicing) would expand fewer nodes than the sequential algorithm on the average. In this section, we show that for a certain type of search space, the average value of the search overhead factor in parallel DFS is less than one. Hence, if the communication overhead is not too large, then on the average, parallel DFS will provide superlinear speedup for this type of search space.

Assumptions

1. The state-space tree has M leaf nodes. Solutions occur only at leaf nodes. The amount of computation needed to generate each leaf node is the same. The number of nodes generated in the tree is proportional to the number of leaf nodes generated. This is a reasonable assumption for search trees in which each node has more than one successor on the average.
2. Both sequential and parallel DFS stop after finding one solution.

3. In parallel DFS, the state-space tree is equally partitioned among p processors; thus, each processor

gets a subtree with M/p leaf nodes.

4. There is at least one solution in the entire tree. (Otherwise, both parallel search and sequential search generate the entire tree without finding a solution, resulting in linear speedup.)

5.There is no information to order the search of the state-space tree; hence, the density of solutions across the unexplored nodes is independent of the order of the search.

6.The solution density r is defined as the probability of the leaf node being a solution. We assume a Bernoulli distribution of solutions; that is, the event of a leaf node being a solution is independent of any other leaf node being a solution. We also assume that r 1.

7. The total number of leaf nodes generated by p processors before one of the processors finds a solution is denoted by Wp. The average number of leaf nodes generated by sequential DFS before a solution is found is given by W. Both W and Wp are less than or equal to M.

Analysis of the Search Overhead Factor

Consider the scenario in which the M leaf nodes are statically divided into p regions, each with K = M/p leaves. Let the density of solutions among the leaves in the i th region be ri. In the parallel algorithm, each processor Pi searches region i independently until a processor finds a solution. In the sequential algorithm, the regions are searched in random order.

Theorem 11.6.1 Let r be the solution density in a region; and assume that the number of leaves K in the region is large. Then, if r > 0, the mean number of leaves generated by a single processor searching the region is 1/r.

Proof: Since we have a Bernoulli distribution, the mean number of trials is given by

Equation 11.5

$$\begin{aligned} \rho + 2\rho(1-\rho) + \cdots + K\rho(1-\rho)^{K-1} &= \frac{1-(1-\rho)^{K+1}}{\rho} - (K+1)(1-\rho)^K, \\ &= \frac{1}{\rho} - (1-\rho)^K\left(\frac{1}{\rho} + K\right). \end{aligned}$$