UNIT –III

PARALLEL ALGORITHMS

Models of parallel computation including PRAM - CRCW, CREW, ERCW, EREW models,

Parallel Random-Access Machines (PRAM) is a model, which is considered for most of the parallel algorithms. Here, multiple processors are attached to a single block of memory. A PRAM model contains −

- A set of similar type of processors.
- All the processors share a common memory unit. Processors can communicate among themselves through the shared memory only.
- A memory access unit (MAU) connects the processors with the single shared memory.

PRAM Architecture

Here, n number of processors can perform independent operations on n number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors.

- To solve this problem, the following constraints have been enforced on PRAM model
- Concurrent Read Concurrent Write (CRCW) − All the processors are allowed to read from or write to the same memory location at the same time.
- Concurrent Read Concurrent Write (CRCW) PRAM: Both simultaneous reads and both simultaneous writes of the same memory cell are allowed.
- Concurrent Read has a clear semantics, whereas Concurrent Write has to be further constrained. There exist several basic sub models.

PRIORITY CRCW:

the processors are assigned fixed distinct priorities and the processor with the highest priority is allowed to complete WRITE.

ARBITRARY CRCW:

one randomly chosen processor is allowed to complete WRITE. The algorithm may make no assumptions about which processor was chosen.

COMMON CRCW:

all processors are allowed to complete WRITE if all the values to be written are equal. Any algorithm for this model has to make sure that this condition is satisfied. If not, the algorithm is illegal, and the machine state will be undefined.

Exclusive Read Exclusive Write (EREW) – Here no two processors are allowed to read from or write to the same memory location at the same time.

Exclusive Read Concurrent Write (ERCW) − Here no two processors are allowed to read from the same memory location at the same time, but are allowed to write to the same memory location at the same time.

Concurrent Read Exclusive Write (CREW) – Here all the processors are allowed to read from the same memory location at the same time, but are not allowed to write to the same memory location at the same time.

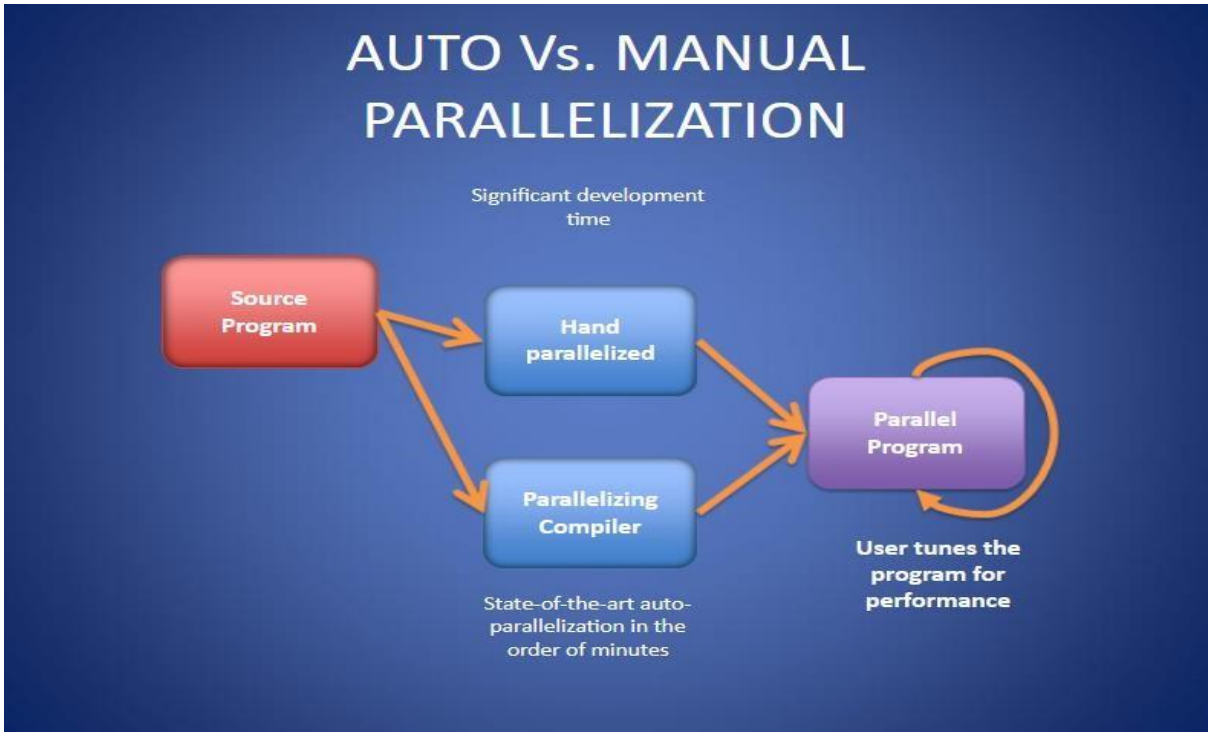
Designing and Analysis of Parallel Algorithms

Automatic vs. Manual Parallelization

Parallelization is the act of designing a computer program or system to process data in parallel. Normally, computer programs compute data serially: they solve one problem, and then the next, then the next.

If a computer program or system is parallelized, it breaks a problem down into smaller pieces to be independently solved simultaneously by discrete computing resources.

Automatic vs. Manual Parallelization



Designing and developing parallel programs has characteristically been a very manual process.

The programmer is typically responsible for both identifying and actually implementing parallelism.

Very often, manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.

For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs.

The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

A parallelizing compiler generally works in two different ways:

- Fully Automatic

The compiler analyses the source code and identifies opportunities for parallelism.

The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.

Loops (do, for) loops are the most frequent target for automatic parallelization.

- Programmer Directed

Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.

May be able to be used in conjunction with some degree of automatic parallelization also.

If you are beginning with an existing serial code and have time or budget constraints,

then automatic parallelization may be the answer.

However, there are several important caveats that apply to automatic parallelization:

- Wrong results may be produced
- Performance may actually degrade
- Much less flexible than manual parallelization
- Limited to a subset (mostly loops) of code
- May actually not parallelize code if the analysis suggests there are inhibitors, or the code is too complex

Parallel algorithms for MIMD machines discussed earlier applies to the manual method of developing parallel codes.

Partitioning

- The partitioning stage of a design is intended to expose opportunities for parallel execution.Focus is on defining a large number of small tasks (fine-grained decomposition).

- A good partition divides both the computation and the data into small pieces. One approach is to focus first on partitioning the data associated with a problem; this is called domain decomposition.
- The alternative approach, termed functional decomposition, decomposes the computation into separate tasks before considering how to partition the data.

These are complementary techniques. Seek to avoid replicating computation and data (may change this later in process)

Communication:

The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing. Issues … Shared memory processing needs additional locking for the data, imposes the overhead of additional processor and bus cycles, and also serializes some portion of the algorithm.

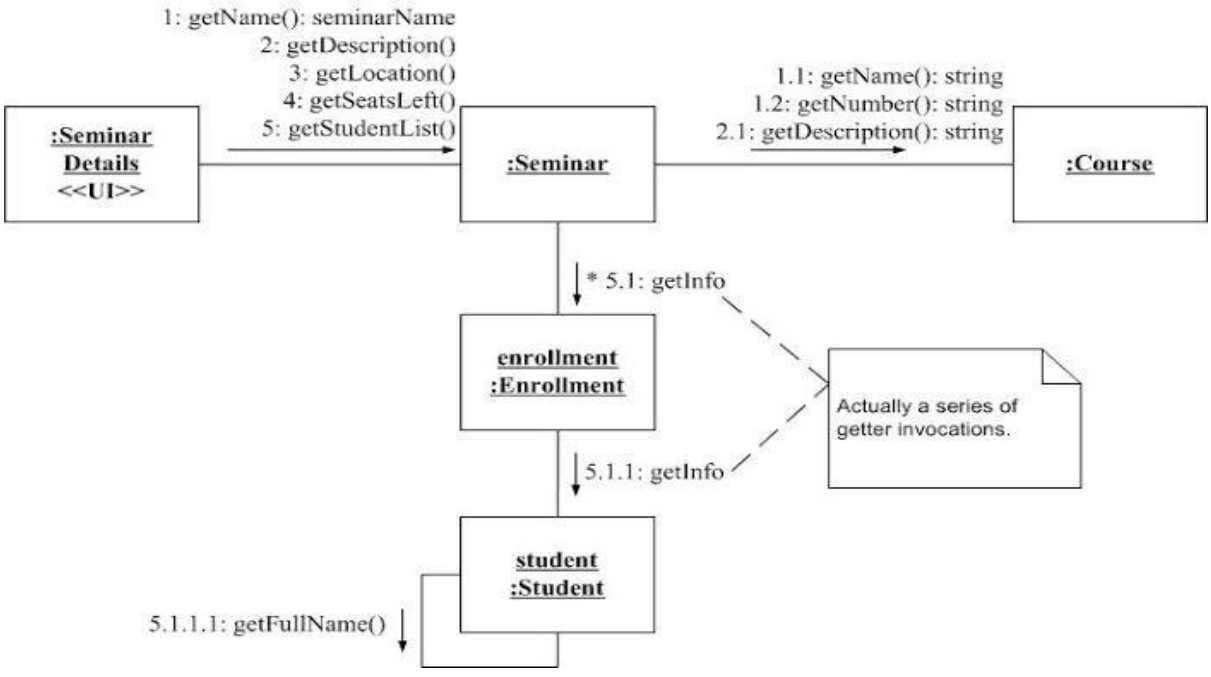
>>> In local communication, each task communicates with a small set of other tasks(neighbours") in contrast, global communication requires each task to communication with many tasks.

>>> In structure communication, a task and its neighbours from a regular structure, such as a tree or grid; in contract, unstructured communication network.

>>> In static communication, the identity of communication partners does not change over time; communication the identity of communication structures may be determined by data computed at runtime and may be highly variable.

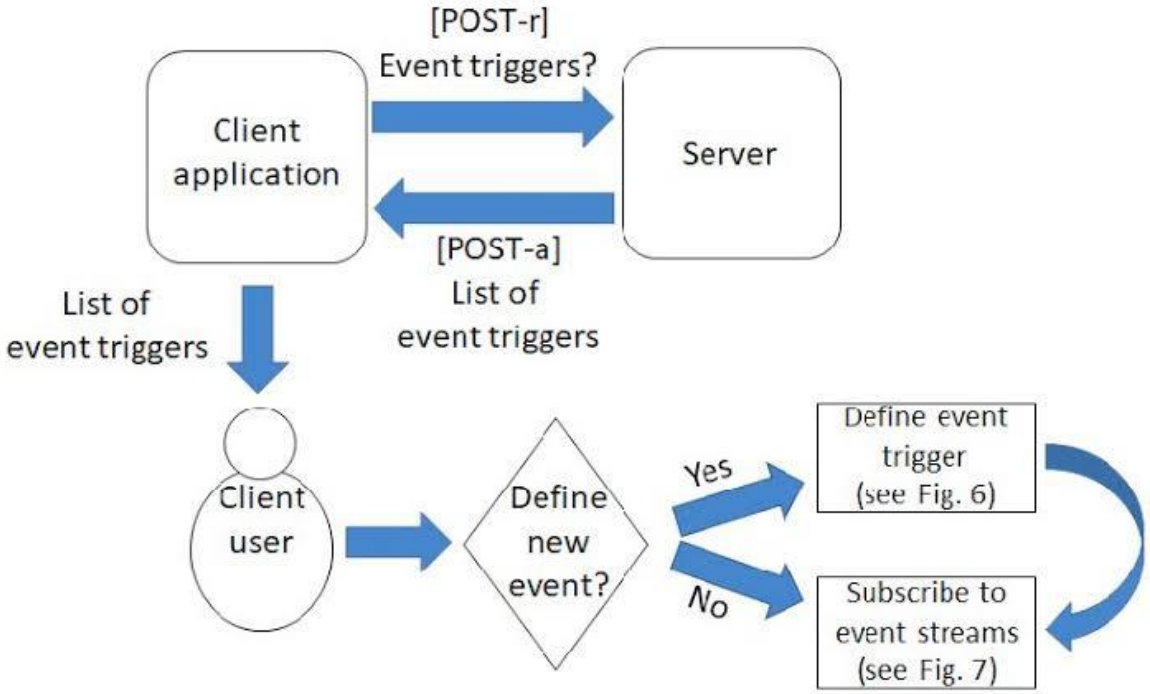
Local Communication:

A local communication structure is obtained when an operation requires data from a small number of other tasks. It is then straightforward to define channels that link the task responsible for performing the operation (the consumer) with the tasks holding the required data (the producers) and to introduce appropriate send and receive operations in the producer and consumer tasks, respectively.



Global Communication:

- A centralized summation algorithm that uses a central manager task (S) to sum N



numbers distributed among N tasks.

- A global communication operation is one in which many tasks must participate. When such operations are implemented, it may not be sufficient simply to identify individual producer/consumer pairs. Such an approach may result in too many communications or may restrict opportunities for concurrent execution.

- Parallel algorithms need to optimize one more resource, the communication between different processors. ... Message passing processing uses channels and message boxes, but this communication adds transfer overhead on the bus, additional memory need for queues and message boxes and latency in the messages.

Synchronization

Critical The critical directive uses to specify a region of code which should execute only one thread at a time. If in the current time a thread is executing inside a critical region and another thread try to reach that region and attempts to perform its command, the last thread will be blocked until the first thread exits that critical region. Furthermore, the optional name permits various and different critical regions to exist the operation, these names perform as global identifiers while different critical regions with the same name are treated as the same region. On the other hand, all critical sections which are unnamed, are treated as the same region. In critical directive, it is illegal to branch into or out of a critical block. In this example, all threads in the team will challenge to execute in parallel, yet, because of the critical construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time.

```
#include

main()

 {

 int x;

 x = 0;

#pragma omp parallel shared(x)

{

#pragma omp critical x = x + 1;

 }/* end of parallel section */

}
```

Atomic:

The Open MP critical section is extremely broad, and there is an overhead every time a thread enters and exits the critical section. Atomic delivers similar mutual exclusion as the above critical pragma, but it only applies to the update of a memory location. An atomic operation has much lower overhead, It depend on the hardware to do the atomic operation.

```
#pragma omp parallel

{

 double tmp, B;

B = drool();

 #pragma omp atomic X += tmp;

 }
```

Barrier:

Barrier is one of the important concept when it comes to synchronization. Basically Barrier provides a point till which all the threads has to wait, once all the threads reach that waiting point then the remaining process will be continued further in the same parallel manner and the process will be repeated. The barriers are classified into two types, namely implicit and explicit. The implicit barriers are generally placed at the end of the block which can be ignored by "nowait" clause placed at the end of pragma directive. While "barrier" clause can be used to place explicit barrier following an if, while, do, switch or label along with a point where no or all threads should wait.

```
#pragma omp parallel
{
int id = omp_get_thread_num();
A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
for (i=0; i < N; i++)
{
C[i] = big_calc3(i,A);
} // implicit barrier
#pragma omp for nowait
for (i=0; i < N; i++)
{
A[id] = big_calc4(id);
} // no implicit barrier due to nowait
} // implicit barrier at the end of parallel region
```

Ordered:

Orded is used for sequential execution (specified using "order" region) untill which threads will be executed concurrently and after the "order" region they will be executed sequentially in the same manner as serial loop.

```
#pragma omp parallel for ordered schedule(dynamic, 4)
for (i=0; i<N; i++);
{
```

```
tmp = foo(i);

#program ordered

Res +=consume(tmp);

}
```

Low Level Synchronization:

Flush Also known as flush set in memory it identifies a point at which the compiler has to ensure that all the threads are viewing a specific object in the same view. All the visible variables will be the part of the flush set if any arguments are not specified in the flush clause. Flush guarantees that all the operations (if they overlap) like read and write will occur before the execution of flush is completed. All the operations will be paused until the flush gets executed. Also, if there are Flushes with overlapping flush, then they cannot be reordered. Flush will force the data to be updated in memory so that every time the thread will used most recent values. Generally, the flushes are active at the entry to and exit from parallel barriers, parallel work sharing (if no wait is not there), ordered and critical region.

Block Synchronization:

Requires all the threads (fundamental means of parallel execution) within a block to be synchronized; while all threads in a grid execute the same kernel function. Threads within a block usually don't complete their tasks simultaneously. [9] To ensure all threads are synchronized at a certain point in an application synchronization barrier, __synchthreads() is implemented. Once one thread reaches the __syncthreads() call, it will wait until all threads have reached it before executing the next instruction. It can be used to ensure memory transfers are completed, all threads reach the end of the loop, or all threads complete their assignments. In the following example, a synchronization barrier is placed after the addition operation to ensure all threads complete the addition operation before continuing.

```
_global__void addition (parameters)

{

some code

Result[ix] = a[ix] + b[ix];

__syncthreads();

}
```

Stream Synchronization:

A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. To ensure all streams are synchronized, a synchronization barrier is used. There are two types of stream synchronization.

Explicit:

Placing the synchronization barrier explicitly, to synchronize tasks such as memory operations. - All streams created on the device are non-blocking; that is, they do not support implicit synchronization with the default NULL stream. Invoking multiple concurrent kernels requires slightly more programming than invoking one kernel at a time. –

Synchronize everything:

• cudaDeviceSynchronize() : Waits until all preceding commands in all streams of all host threads have completed i.e. blocks host until all issued CUDA calls are complete - Synchronize w.r.t. a specific stream

• cudaStreamSynchronize(streamid) : Takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

Synchronize using Events

1. Create specific 'Events', within streams, to use for synchronization.

2. CudaEventRecord(event, streamid) : takes in stream and record the parameter event. Resources associated with event is locked till devices that uses it are completed.

3. CudaEventSynchronizee(event): similar to cudaStreamSynchronize , it waits for devices that uses event that was recorded to be complete.

4. CudaStreamWaitEvent(stream, event) : takes a stream and an event as parameters and makes all the commands added to the given stream after the call to cudaStreamWaitEvent() delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to cudaStreamWaitEvent() wait on the event.

5. CudaStreamQuery() : provides applications with a way to know if all preceding commands in a stream have completed

Dependency Analysis

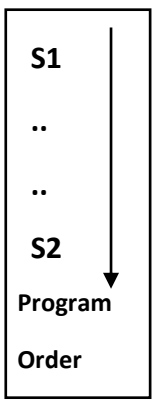
- Dependency analysis is concerned with detecting the presence and type of dependency between tasks that prevent tasks from being independent and from running in parallel on different processors and can be applied to tasks of any grain size*.
  - Represented graphically as task dependency graphs.
- Dependencies between tasks can be 1- algorithm/program related or 2- hardware resource/architecture related.
- Algorithm/program Task Dependencies:
  - Data Dependence:
    - True Data or Flow Dependence
    - Name Dependence:
      - Anti-dependence
      - Output (or write) dependence
  - Control Dependence
- Hardware/Architecture Resource Dependence

*Task Grain Size: Amount of computation in a task

Data & Name Dependence

Assume task S2 follows task S1 in sequential program order

1 True Data or Flow Dependence: Task S2 is data dependent on task S1 if an execution path exists from S1 to S2 and if at least one output variable of S1 feeds in as an input operand used by S2



Represented by S1 ⟶ S2 in task dependency graphs

2 Anti-dependence: Task S2 is anti-dependent on S1, if S2 follows S1 in program order and if the output of S2 overlaps the input of S1

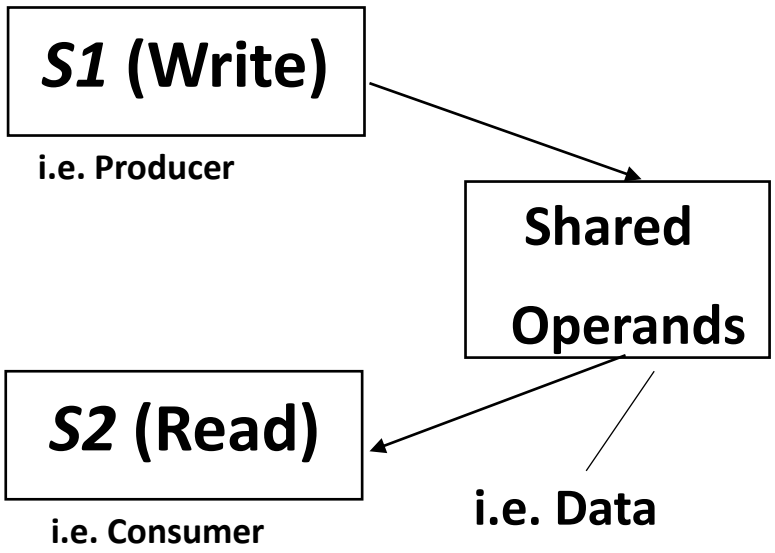
Represented by S1 ⟶ S2 in dependency graphs

3 Output dependence: Two tasks S1, S2 are output dependent if they produce the same output variables

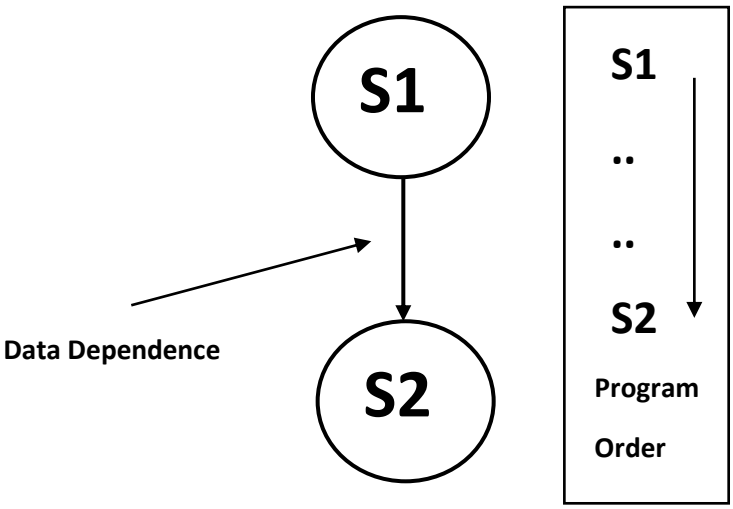
Represented by S1 ⟶ S2 in task dependency graphs

(True) Data (or Flow) Dependence

- Assume task S2 follows task S1 in sequential program order
- Task S1 produces one or more results used by task S2,
  - Then task S2 is said to be data dependent on task S1
- Changing the relative execution order of tasks S1, S2 in the parallel program violates this data dependence and results in incorrect execution.



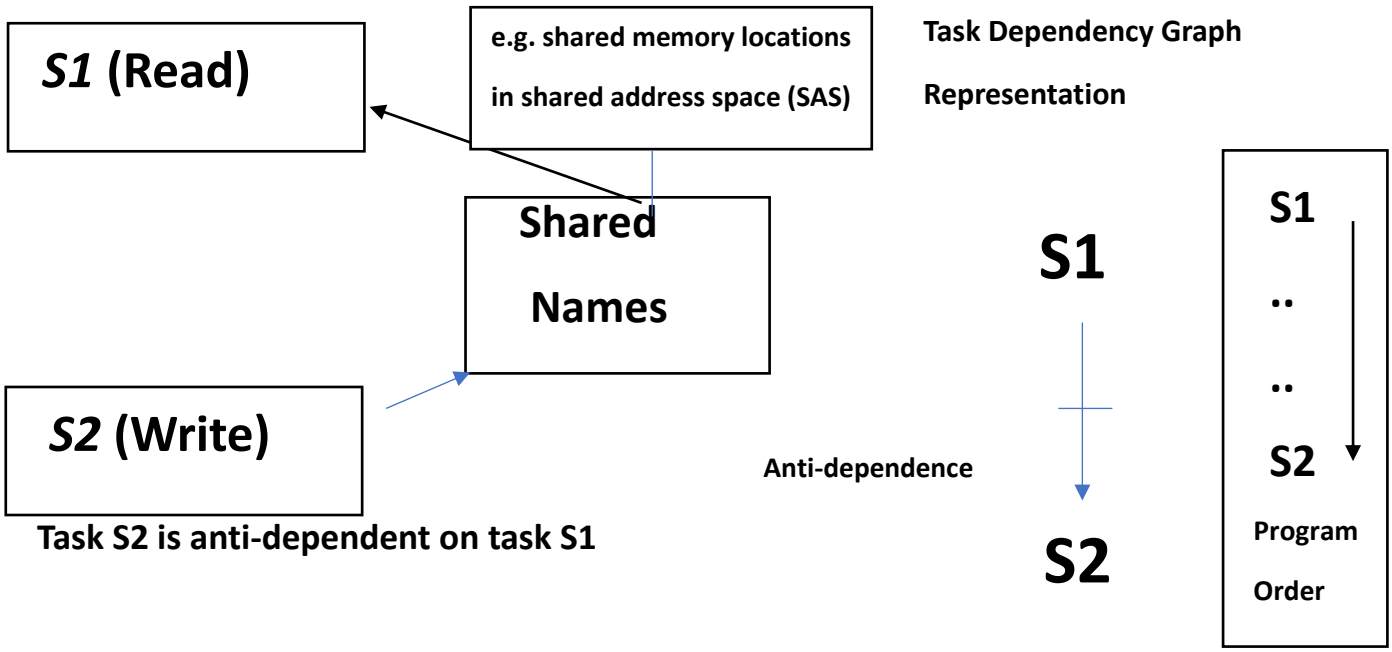
Task Dependency Graph Representation



Task S2 is data dependent on task S1

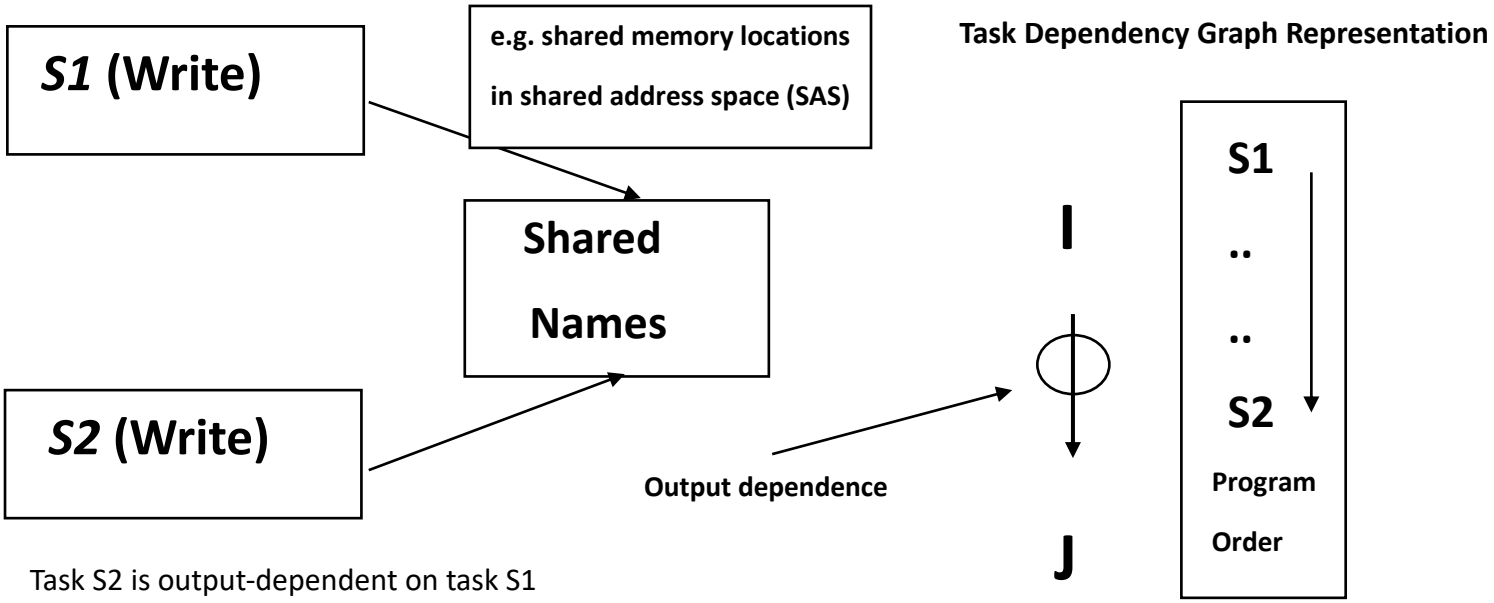
Name Dependence Classification: Anti-Dependence

- Assume task S2 follows task S1 in sequential program order
- Task S1 reads one or more values from one or more names (registers or memory locations)
- Task S2 writes one or more values to the same names (same registers or memory locations read by S1)
  - Then task S2 is said to be anti-dependent on task S1
- Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.



Name Dependence Classification: Output (or Write) Dependence

- Assume task S2 follows task S1 in sequential program order
- Both tasks S1, S2 write to the same a name or names (same registers or memory locations)
  - Then task S2 is said to be output-dependent on task S1
- Changing the relative execution order of tasks S1, S2 in the parallel program violates this name dependence and may result in incorrect execution.



Task Dependency Graph Representation

Task S2 is output-dependent on task S1

Dependency Graph Example

Here assume each instruction is treated as a task

S1: Load R1, A / R1 ¬Memory(A) /

S2: Add R2, R1 / R2 ¬ R1 + R2 /

S3: Move R1, R3 / R1¬ R3 /

S4: Store B, R1 /Memory(B) ¬ R1 /

True Date Dependence:

(S1, S2)(S3, S4)

S1⟶ S2, S3 ⟶S4

Output Dependence:

(S1, S3)

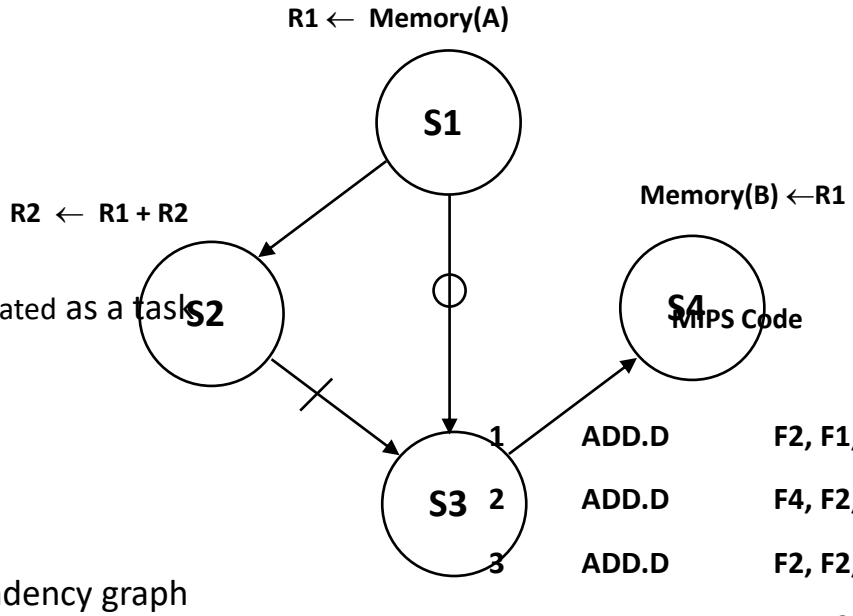
Here assume each instruction is treated as a task

i.e. S1 ⟶ S3

Anti-dependence:

(S2, S3)

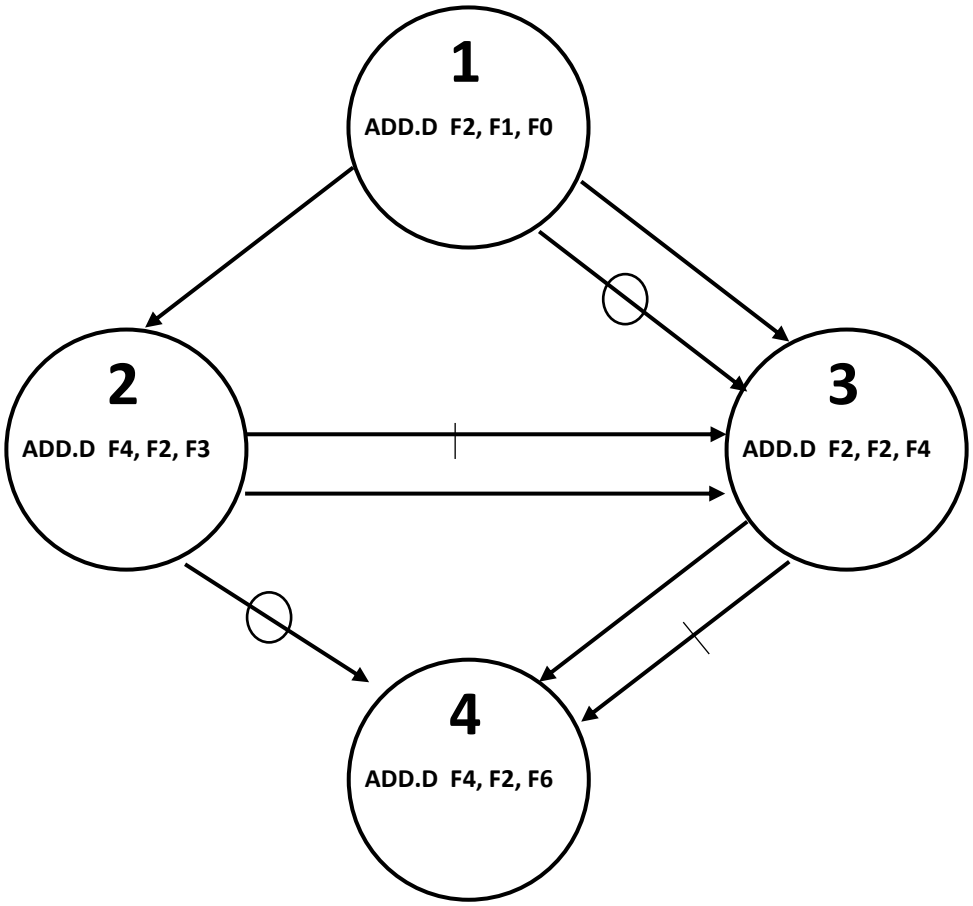
i.e. S2 ⟶ S3



Dependency graph

MIPS Code

```
1   ADD.D   F2, F1, F0
2   ADD.D   F4, F2, F3
3   ADD.D   F2, F2, F4
4   ADD.D   F4, F2, F6
```



True Date Dependence:

(1, 2) (1, 3) (2, 3) (3, 4)

i.e. 1 → 2 1 → 3

2 → 3 3 → 4

Output Dependence:

(1, 3) (2, 4)

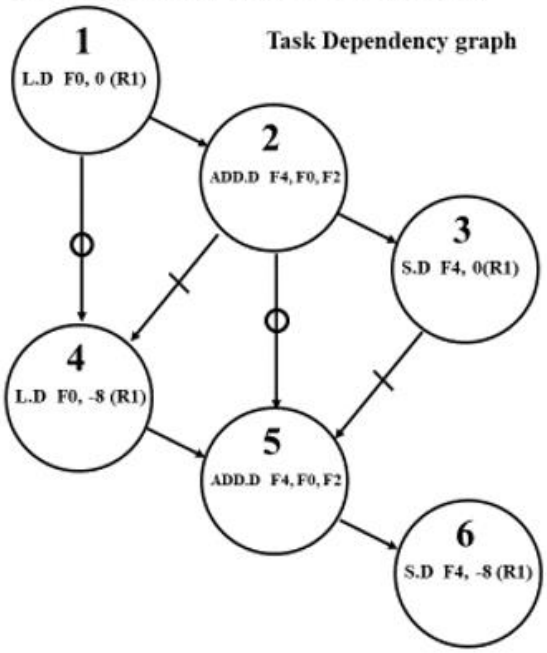
i.e. 1 → 3 2 → 4

Anti-dependence:

(2, 3) (3, 4)

i.e. 2 → 3 3 → 4

Here assume each instruction is treated as a task



MIPS Code

1	L.D	F0, 0 (R1)
2	ADD.D	F4, F0, F2
3	S.D	F4, 0(R1)
4	L.D	F0, -8(R1)
5	ADD.D	F4, F0, F2
6	S.D	F4, -8(R1)

True Date Dependence:
(1, 2) (2, 3) (4, 5) (5, 6)
i.e. 1 → 2 1 → 3
4 → 5 5 → 6

Output Dependence:
(1, 4) (2, 5)
i.e. 1 → 4 2 → 5

Anti-dependence:
(2, 4) (3, 5)
i.e. 2 → 4 3 → 5

LOAD BALANCING IN PARALLEL ALGORTHIMS

In computing, **load balancing** refers to the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient. Load balancing techniques can optimize the response time for each task, avoiding unevenly overloading computer nodes while other computer nodes are left idle.

Two main approaches exist: static algorithms, which do not take into account the state of the different machines, and dynamic algorithms, which are usually more general and more efficient, but require exchanges of information between the different computing units, at the risk of a loss of efficiency.

Static and dynamic algorithms

Static

A load balancing algorithm is "static" when it does not take into account the state of the system for the distribution of tasks. Thereby, the system state includes measures such as the load level (and sometimes even overload) of certain processors. Instead, assumptions on the overall system are made beforehand, such as the arrival times and resource requirements of incoming tasks. In addition, the number of processors, their respective power and communication speeds are known. Therefore, static load balancing aims to associate a known set of tasks with the available processors in order to minimize a certain performance function. The trick lies in the concept of this performance function.

Static load balancing techniques are commonly centralized around a router, or master which distributes the loads and optimizes the performance function. This minimization can take into account information related to the tasks to be distributed and derive an expected execution time.

The advantage of static algorithms is that they are easy to set up and extremely efficient in the case of fairly regular tasks (such as processing HTTP requests from a website). However, there is still some statistical variance in the assignment of tasks which can lead to overloading of some computing units.

Dynamic

Unlike static load distribution algorithms, dynamic algorithms take into account the current load of each of the computing units (also called nodes) in the system. In this approach, tasks can be moved dynamically from an overloaded node to an underloaded node in order to receive faster processing. While these algorithms are much more complicated to design, they can produce excellent results, in particular, when the execution time varies greatly from one task to another.

In dynamic load balancing the architecture can be more modular since it is not mandatory to have a specific node dedicated to the distribution of work. When tasks are uniquely assigned to a processor according to its state at a given moment, it is unique assignment. If, on the other hand, the tasks can be permanently redistributed according to the state of the system and its evolution, this is called dynamic assignment. Obviously, a load balancing algorithm that requires too much communication in order to reach its decisions runs the risk of slowing down the resolution of the overall problem.

There are numerous techniques and algorithms that can be used to intelligently load balance client access requests across server pools. The technique chosen will depend on the type of service or application being served and the status of the network and servers at the time of the request. The current level of requests to the load balancers often determines which method is used. When the load is low then one of the simple load balancing methods will suffice. In times of high load, the more complex methods are used to ensure an even distribution of requests.

Load Balancing techniques:

Round Robin

Round-robin load balancing is one of the simplest and most used load balancing algorithms. Client requests are distributed to application servers in rotation. For example, if you have three application servers: the first client request to the first application server in the list, the second client request to the second application server, the third client request to the third application server, the fourth to the first application server and so on.
This load balancing algorithm does not take into consideration the characteristics of the application servers i.e. it assumes that all application servers are the same with the same availability, computing and load handling characteristics.

Weighted Round Robin

Weighted Round Robin builds on the simple Round-robin load balancing algorithm to account for differing application server characteristics. The administrator assigns a weight to each application server based on criteria of their choosing to demonstrate the application servers' traffic-handling capability.

Example: - If application server #1 is twice as powerful as application server #2 (and application server #3), application server #1 is provisioned with a higher weight and application server #2 and #3 get the same weight. If there five (5) sequential client requests, the first two (2) go to application server #1, the third (3) goes to application server #2, the fourth (4) to application server #3 and the fifth (5) to application server #1.

Least Connection: Least Connection load balancing is a dynamic load balancing algorithm where client requests are distributed to the application server with the least number of active connections at the time the client request is received. In cases where application servers have similar specifications, an application server may be overloaded due to longer lived connections; this algorithm takes the active connection load into consideration.

Weighted Least Connection: Weighted Least Connection builds on the Least Connection load balancing algorithm to account for differing application server characteristics. The administrator assigns a weight to each application server based on criteria of their choosing to demonstrate the application servers' traffic-handling capability. The Loadmaster is making the load balancing criteria based on active connections and application server weighting.

Resource Based (Adaptive): Resource Based (Adaptive) is a load balancing algorithm requires an agent to be installed on the application server that reports on its current load to the load balancer. The installed agent monitors the application server's availability status and resources. The load balancer queries the output from the agent to aid in load balancing decisions.

Resource Based (SDN Adaptive): SDN Adaptive is a load balancing algorithm that combines knowledge from Layers 2, 3, 4 and 7 and input from an SDN Controller to make more optimized traffic distribution decisions. This allows information about the status of the servers, the status of the applications running on them, the health of the network infrastructure, and the level of congestion on the network to all play a part in the load balancing decision making.

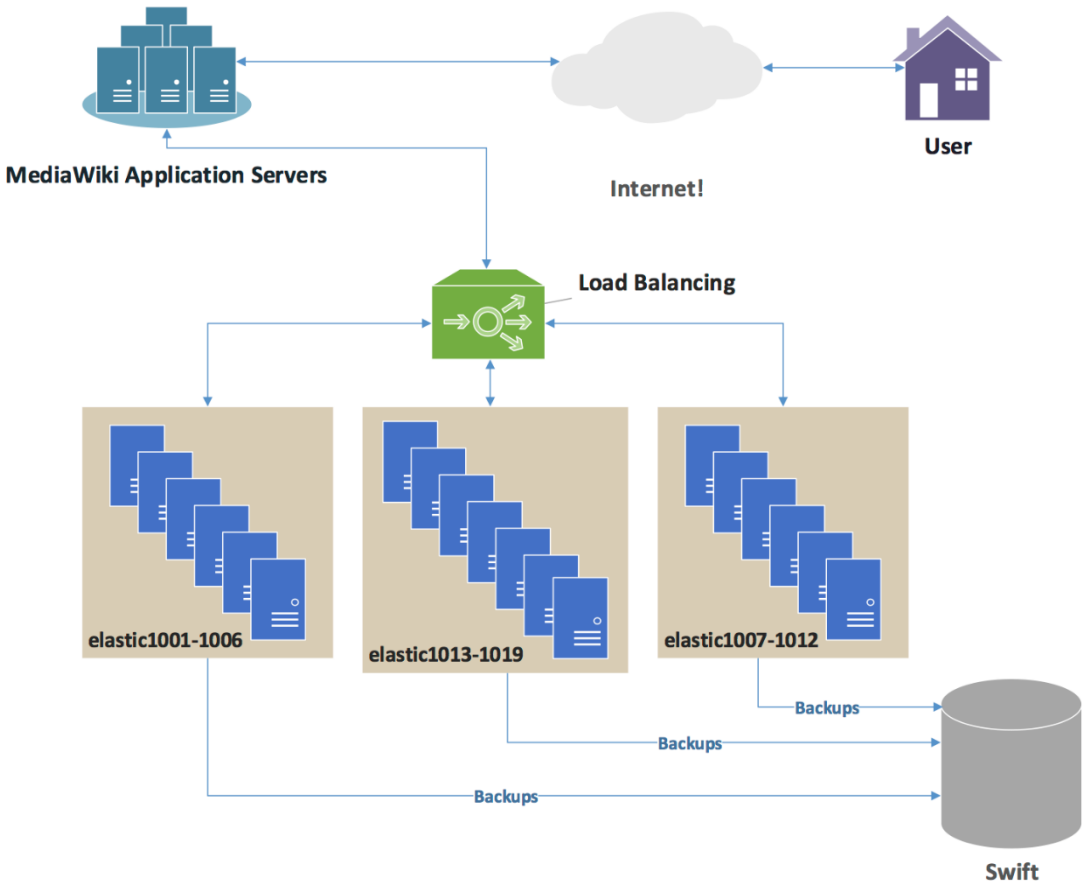
Fixed Weighting: Fixed Weighting is a load balancing algorithm where the administrator assigns a weight to each application server based on criteria of their choosing to demonstrate the application servers traffic-handling capability. The application server with the highest weigh will receive all of the traffic. If the application server with the highest weight fails, all traffic will be directed to the next highest weight application server.

Weighted Response Time: Weighted Response Time is a load balancing algorithm where the response times of the application servers determines which application server receives the next request. The application server response time to a health check is used to calculate the application server weights. The application server that is responding the fastest receives the next request.

Source IP Hash: Source IP hash load balancing algorithm that combines source and destination IP addresses of the client and server to generate a unique hash key. The key is

used to allocate the client to a particular server. As the key can be regenerated if the session is broken, the client request is directed to the same server it was using previously. This is useful if it's important that a client should connect to a session that is still active after a disconnection.

URL Hash: URL Hash is a load balancing algorithm to distribute writes evenly across multiple sites and sends all reads to the site owning the object.



GRANULARITY

In parallel computing, **granularity** (or grain size) of a task is a measure of the amount of work (or computation) which is performed by that task.

Another definition of granularity takes into account the communication overhead between multiple processors or processing elements. It defines granularity as the ratio of computation time to communication time, wherein, computation time is the time required to perform the computation of a task and communication time is the time required to exchange data between processors.

If Tcomp is the computation time and T comm denotes the communication time, then the Granularity G of a task can be calculated as

G=Tcomp / Tcomm

Granularity is usually measured in terms of the number of instructions executed in a particular task. Alternately, granularity can also be specified in terms of the execution time of a program, combining the computation time and communication time.

Types of parallelism

Depending on the amount of work which is performed by a parallel task, parallelism can be classified into three categories: fine-grained, medium-grained and coarse-grained parallelism.

Fine-grained parallelism

In fine-grained parallelism, a program is broken down to a large number of small tasks. These tasks are assigned individually to many processors. The amount of work associated with a parallel task is low and the work is evenly distributed among the processors. Hence, fine-grained parallelism facilitates load balancing.

As each task processes less data, the number of processors required to perform the complete processing is high. This in turn, increases the communication and synchronization overhead.

Fine-grained parallelism is best exploited in architectures which support fast communication. Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.

It is difficult for programmers to detect parallelism in a program, therefore, it is usually the compilers' responsibility to detect fine-grained parallelism.

An example of a fine-grained system (from outside the parallel computing domain) is the system of neurons in our brain.

Connection Machine (CM-2) and **J-Machine** are examples of fine-grain parallel computers that have grain size in the range of 4-5 μs.

Coarse-grained parallelism:

In coarse-grained parallelism, a program is split into large tasks. Due to this, a large amount of computation takes place in processors. This might result in load imbalance, wherein certain tasks process the bulk of the data while others might be idle. Further, coarse-grained parallelism fails to exploit the parallelism in the program as most of the computation is performed sequentially on a processor. The advantage of this type of parallelism is low communication and synchronization overhead.

Message-passing architecture takes a long time to communicate data among processes which makes it suitable for coarse-grained parallelism.

Cray Y-MP is an example of coarse-grained parallel computer which has a grain size of about 20s.

Medium-grained parallelism

Medium-grained parallelism is used relatively to fine-grained and coarse-grained parallelism. Medium-grained parallelism is a compromise between fine-grained and coarse-grained parallelism, where we have task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism. Most general-purpose parallel computers fall in this category.

Intel iPSC is an example of medium-grained parallel computer which has a grain size of about 10ms.

Consider a 10*10 image that needs to be processed, given that, processing of the 100 pixels is independent of each other.

Fine-grained parallelism: Assume there are 100 processors that are responsible for processing the 10*10 image. Ignoring the communication overhead, the 100 processors can process the 10*10 image in 1 clock cycle. Each processor is working on 1 pixel of the image

and then communicates the output to other processors. This is an example of fine-grained parallelism.

Medium-grained parallelism: Consider that there are 25 processors processing the 10*10 image. The processing of the image will now take 4 clock cycles. This is an example of medium-grained parallelism.

Coarse-grained parallelism: Further, if we reduce the processors to 2, then the processing will take 50 clock cycles. Each processor needs to process 50 elements which increases the computation time, but the communication overhead decreases as the number of processors which share data decreases. This case illustrates coarse-grained parallelism.

Fine-grain: Pseudocode for 100 processors Medium-grain: Pseudocode for 25 processors Coarse-grain : Pseudocode for 2 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute element 1; break;
    case 2: Compute element 2; break;
    case 3: Compute element 3; break;
    .
    .
    .
    .
    case 100: Compute element 100;
            break;
  }
}
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-4; break;
    case 2: Compute elements 5-8; break;
    case 3: Compute elements 9-12; break;
    .
    .
    case 25: Compute elements 97-100;
           break;
  }
}
void main()
```

```
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-50;
          break;
    case 2: Compute elements 51-100;
          break;
  }
}
```

Computation time - 1 clock cycle computation time - 4 clock cyclesComputation time - 50 clock cycles

Levels of parallelism

Granularity is closely tied to the level of processing. A program can be broken down into 4 levels of parallelism -

Instruction level.

Loop level

Sub-routine level and

Program-level

The highest amount of parallelism is achieved at instruction level, followed by loop-level parallelism. At instruction and loop level, fine-grained parallelism is achieved. Typical grain size at instruction-level is 20 instructions, while the grain-size at loop-level is 500 instructions.

At the sub-routine (or procedure) level the grain size is typically a few thousand instructions. Medium-grained parallelism is achieved at sub-routine level.

At program-level, parallel execution of programs takes place. Granularity can be in the range of tens of thousands of instructions. Coarse-grained parallelism is used at this level.

The below table shows the relationship between levels of parallelism, grain size and degree of parallelism

Levels	Grain Size	Parallelism
Instruction level	Fine	Highest
Loop level	Fine	Moderate

Sub-routine level	Medium	Moderate
Program level	Coarse	Least

Impact of granularity on performance

Granularity affects the performance of parallel computers. Using fine grains or small tasks results in more parallelism and hence increases the speedup. However, synchronization overhead, scheduling strategies etc. can negatively impact the performance of fine-grained tasks. Increasing parallelism alone cannot give the best performance.

In order to reduce the communication overhead, granularity can be increased. Coarse grained tasks have less communication overhead, but they often cause load imbalance. Hence optimal performance is achieved between the two extremes of fine-grained and coarse-grained parallelism.

Various studies have proposed their solution to help determine the best granularity to aid parallel processing. Finding the best grain size, depends on a number of factors and varies greatly from problem-to-problem.

INPUT/OUTPUT LIMITS

Input/Output in Parallel and Distributed Computer Systems has attracted increasing attention over the last few years, as it has become apparent that input/output performance, rather than CPU performance, may be the key limiting factor in the performance of future systems.

This I/O bottleneck is caused by the increasing speed mismatch between processing units and storage devices, the use of multiple processors operating simultaneously in parallel and distributed systems, and by the increasing I/O demands of new classes of applications, like multimedia. It is also important to note that, to varying degrees, the I/O bottleneck exists at multiple levels of the memory hierarchy.

All indications are that the I/O bottleneck will be with us for some time to come and is likely to increase in importance. Input/Output in Parallel and Distributed Computer Systems is based on papers presented at the 1994 and 1995 IOPADS workshops held in conjunction with the International Parallel Processing Symposium.

This book is divided into three parts. Part I, the Introduction, contains four invited chapters which provide a tutorial survey of I/O issues in parallel and distributed systems.

The chapters in Parts II and III contain selected research papers from the 1994 and 1995 IOPADS workshops; many of these papers have been substantially revised and updated for inclusion in this volume.

Part II collects the papers from both years which deal with various aspects of system software, and Part III addresses architectural issues.

Input/Output in Parallel and Distributed Computer Systems is suitable as a secondary text for graduate level courses in computer architecture, software engineering, and multimedia systems, and as a reference for researchers and practitioners in industry.

Abstract

We sketch the reasons for the I/O bottleneck in parallel and distributed systems, pointing out that it can be viewed as a special case of a general bottleneck that arises at all levels of the memory hierarchy. We argue that because of its severity, the I/O bottleneck deserves systematic attention at all levels of system design. We then present a survey of the issues raised by the I/O bottleneck in five key areas of parallel and distributed systems: applications, algorithms, compilers, operating systems and architecture. Finally, we address some of the trends we observe emerging in new paradigms of parallel and distributed computing: the convergence of networking and I/O, I/O for massively distributed “global information systems” such as the World Wide Web, and I/O for mobile computing and wireless

communications. These considerations suggest exciting new research directions in I/O for parallel and distributed systems in the years to come.

Input/Output.

Closely connected to the problem seriality appears the seriality for input and output, needed by the problem. The amount of input/output is primarily dependent on the type of this problem. The time required for it works in the same direction of limiting performance, but by decreasing with the number of processing modules after an optimal value. Three models are treated here: a linear array of processor modules, a rectangular, and a square configuration.

Following are the limitations of input-output analysis:

- Constancy of **Input** Coefficient Assumption Unrealistic: ADVERTISEMENTS: ...
- Factor Substitution Possible: ...
- Rigid Model: ...
- Restrictive Model: ...
- Difficulty in Final Demand: ...
- Quantity of **Inputs** not Constant: ...
- Solution of Equations Difficult:

Input-Output Table:

The input-output accounting of national income is presented in an input-output table which is based on a 'transactions matrix'. A transactions matrix shows how the total output of one industry is distributed to all other industries as inputs and for final demand.

A set of m n quantities or values arranged in m rows and n columns in a rectangular or square form is a matrix. That is why an input- output table is often called input-output matrix. The columns and rows of an input-output table 'provide industrial breakdowns of the final expenditures and income payments that enter into the national income accounts.

A simple input-output matrix of an economy is shown in Table 7. Its rows show the amount of each industry's output sold to every other industry and to final buyers. The columns show the amount of each industry's inputs bought from every other industry, and from imports

and factor services, known as primary inputs because they are not produced by the industries in the country.

Table 7 : Input-Output Transaction Matrix

(Rs Crores)

Purchasing Sectors → / Selling Sectors ↓	Inputs to: Agriculture	Manufacturing	Others	Final Demand (X+K+G+C)	Total Gross Output
	1	2	3	4	5
Agriculture	-	15	5	22	42
Manufacturing	12	-	17	16	45
Others	8	12	-	30	50
Inports	7	5	8	7	27
Primary inputs	15	13	20	-	48
Total Gross Input	*42*	*45*	*50*	*75*	*212*

In this table, the total gross output of the agriculture sector of the economy is set in the first row (to be read horizontally). It consists of Rs. 15 crores to the manufacturing sector, Rs. 5 crores to the other sectors, and Rs. 22 crores to satisfy the final demand which comprises exports (X), capital (K), government (G) and personal consumption (C).

Limitations of Input-Output

1. Constancy of Input Coefficient Assumption Unrealistic:

The input-output analysis has its shortcomings. Its framework rests on the assumption of constancy of input co-efficient of production. It tells us nothing as to how technical coefficients would change with changed conditions.

Again some industries may have identical capital structures some may have heavy capital requirements while others may use no capital. Such variations in the use of techniques of production make the assumption of constant coefficients of production unrealistic.

2. Factor Substitution Possible:

This assumption of fixed coefficients of production ignores the possibility of factor substitution. There is always the possibility of some substitutions even in a short period, while substitution possibilities are likely to be relatively greater over a longer period.

3. Rigid Model:

The rigidity of the input-output model cannot reflect such phenomena as bottlenecks, increasing costs, etc.

4. Restrictive Model:

The input-output model is severely simplified and restricted as it lays exclusive emphasis on the production side for the economy. It does not tell us why the inputs and outputs are of a particular pattern in the economy.

5. Difficulty in Final Demand:

Another difficulty arises in the case of "final demand" or "bill of goods." In this analysis, the purchases by the government and consumers are taken as given and treated as a specific bill of goods. Final demand is regarded as an independent variable. It might, therefore, fail to utilize all the factors proportionately or need more than their available supply. Assuming constancy of co-efficiency of production, the analysis is not in a position to solve this difficulty.

6. Quantity of Inputs not Constant:

This analysis operates on the basis of a fixed quantity of an input for the production of per unit of output. As factors are mostly indivisible, the increases in outputs are not expected to be in proportion to the increases in inputs.

7. Solution of Equations Difficult:

The input-output model works on equations which cannot be solved easily. First, the model of equations is prepared and then large numbers of data are collected. Equations require thorough knowledge of higher mathematics and even the collection of data is not so easy.

This makes the construction of input-output model difficult.

Cost Modelling for Parallel Programming

1. Sequential algorithms can be sped up by running them on faster machines
2. The whole algorithm will speed up
3. Parallel algorithms on parallel architectures benefit from more tuning parameters than

just overall speed of the machine

- Memory usage
- Thread management
- Data management

There are also limitations to the amount of parallelism that can be introduced to an

- algorithm
- Amdahl's Law

Inherent parallelism
Restructuring a parallel algorithm in order to tune it is not straightforward.
The

- fundamental aim of this research is to help programmers perform this restructuring
- Extensive research has produced a large body of information regarding parallelprogramming, algorithms and architectures
- Less clear is how to integrate these – which algorithms suit which architectures
- The investment of programming effort for any combination of algorithm andarchitecture is very high. So it is important to make the correct decision at the outset
- That decision needs to be based on more than just data/task shorthand and intuition
    - Aim is to provide a more concrete foundation to support the decision making process.
- Restructuring a parallel algorithm in order to tune it is not straightforward.

There is a large design space

Multicores – task & data parallelism

1. FPGAs – task & data parallelism
2. CPU clusters – task & data
3. GPUs – data (& task?)

- Too expensive in terms of time & effort to code these up to compare them, need acost model to provide a comparison on which to base a decision

- The cost model can also help in terms of considering development time e.g.:CPU cluster will result in 15% speed up & will require 100 hours of developmenttime GPU will result in 25% speed up & will require 500 hours of development time
- If a 15% performance gain is sufficient for the programmer's needs then CPU cluster may be best solution.

Cost Models

- A cost model expresses the execution time, memory consumption or power consumption of an algorithm as a function of relevant parameters
- It can be used to predict the performance of an algorithm based on those parameters

For example consider the following:

```
int sourceArray[i]; int resArray[i]; int j = 42; for
(k = 0; k < i; k++) if sourceArray[k] < j
resArray[k] = 1;
```

The performance of a parallel algorithm to carry out this task could potentially depend on the following parameters:

1. Number of comparisons
2. Number of threads
3. Number and type of memory accesses
4. Cache behaviour

Assuming off-chip memory is used, the cost model could look like the following: Where:

$$f(x) = \frac{logx((Ax+B)+\frac{x}{y}+(Cx))}{D}$$

x = number of operations y =
number of threads

A & B = constants relating to the number of operations C = constant relating to the number of off-chip memory accesses

D = constant relating to caching behaviour f(x) = gives the predicted performance time in cycles

These constants are known and will vary depending on the architecture used and the structure of the algorithm.

Cost models can assist both in:

1. Choosing the architecture
2. Structuring the algorithm.

They can be used by the programmer to make key decisions based on empirical data.

There is a trade off between the usability and accuracy of the cost model:

The more accurate the model, the more complex it will be to use

- The cost model performance prediction may vary from the actual performance

A percentage measure of accuracy is calculated as:

Where:

A = actual performance P = predicted performance.

For example:

$$100(\frac{A-P}{A})$$

Actual execution time (in cycles) = 105,000

Predicted execution time = 100,000

Accuracy = 4.76%

Predicted execution time = 110,000

Accuracy = -4.76%

- Work at Bayreuth and Chemnitz has concluded that a model which produces an accuracy of the actual time plus or minus 10% is useful

Using a Cost Model in Parallel Programming

Parallel programming encompasses a wide range of architectures, from single nodeswith a few cores to supercomputers

The construction & implementation of a supercomputer can cost $100millions

This makes it important to manage the runtime of applications on such a platform

A parallel application running in an environment such as a multi-node CPU cluster ora supercomputer will consist of many **tasks**.

In this context a task is a piece of work which can run on one or more processors

A task is **not a thread** but may use a collection of threads Task examples:

Execute a specific algorithm (sum reduction of an array)

Update a display

Gather information from remote sensors

A function will exist relating to the task which can be used to calculate how long the task will take to complete dependant on how many processors are allocated to it e.g.

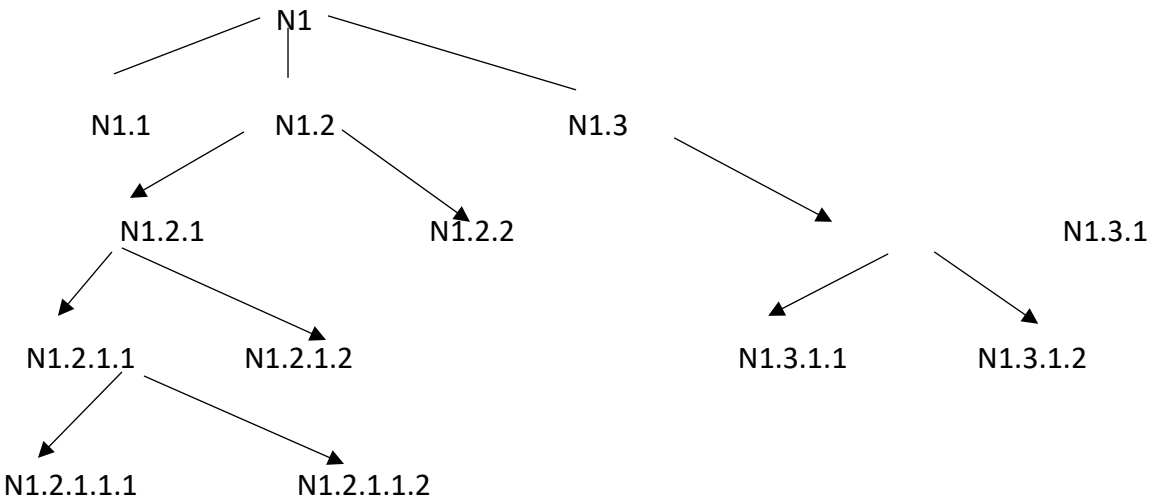
$$f(x) = \frac{A}{x}$$

In this example A=100

x = the number of processors allocated

- The parallel application can be represented as a tree
- Each node in the tree will have processors available to it which will allow theexecution of specific tasks
- A key aspect of structuring a parallel application will include considering the partitionof such tasks, with attention being paid to:

    1. Data dependencies
    2. Communication costs
    3. Processing requirements

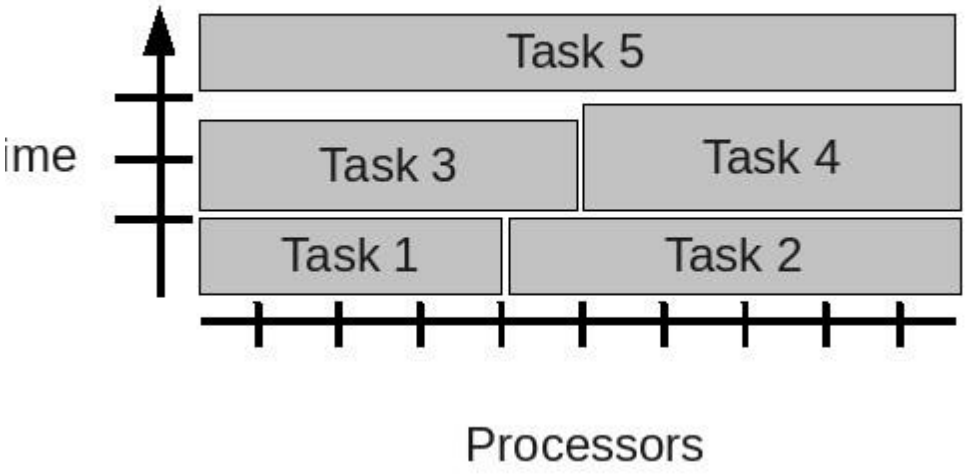


If a cost model is available for each task then it should be possible to predict how long each will take to run

Task Number	Time Units Required
1	40
2	60
3	75

4	100
5	100

These predictions are based on the assumption that one processor is used for 1 task



Development of Cost Models

Theoretical development is based on an analysis of the characteristics of the algorithm and the hardware that it is intended to be run on. For example, the manufacturer of the hardware may produce documentation detailing how long certain operations should take on their hardware e.g.:

- Accessing global memory will take between 400-600 cycles
- Accessing shared memory will take 4 cycles
- Adding two integers will take 4 cycles
- Multiplying two integers will take 16 cycles

The work done by Hong & Kim and Kothapalli et al takes this approach.

Measurement based development focuses on developing benchmarks for standard application operations and measuring how long these take to run. Such operations could include:

- Addition of two integers/floats
- Multiplication of two integers/floats
- Comparing two integers/floats
- Read from memory
- Write to memory
- Transferring data between devices
- Once the measurements are captured, models are then derived using curve fitting

int z, x = 42, y = 24; start_time = clock() loop
n times z += x + y
stop_time = clock() duration = stop_time start_ time.

The Performance Analysis and Tuning

Performance Analysis:

Abstract:

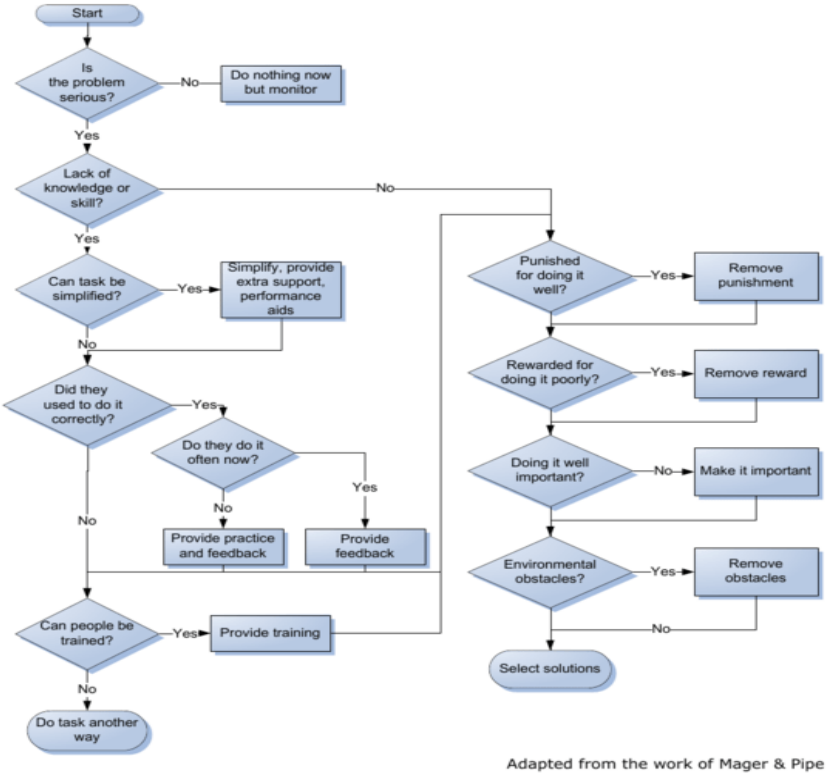
The implementation of client parallel programs can be very dicult, and numerous computer-based tools have been developed to help with the task of performance analysis and tuning. We describe the initial steps in the design of such a tool. Our aim is to show how its design has evolved out of an explicit attempt to understand the human factors issues in parallel program implementation. By adopting this approach, we seek to look beyond the current fashion for graphical user interfaces and data visualization techniques. Our emphasis is on gaining a better understanding of the problems inherent in program analysis and tuning and using this as a foundation for developing more usable tools.

Introduction:

Parallel program performance analysis and tuning is concerned with achieving ecientutilisation of system resources. One common technique is to collect trace data and then analyse it for possible causes of poor performance. The program source code is then modied in the light of the analysis. In practice, tuning is much more dicult than this.

Performance Analysis Flowchart

The performance analysis flowchart developed by Mager and Pipe is a systematic way of reviewing information and identifying potential solutions. It is particularly useful for distinguishing between training and non-training solutions.



Adapted from the work of Mager & Pipe

Performance tuning:

Performance tuning is the improvement of system performance. Typically, in computer systems, the motivation for such activity is called a performance problem, which can be either real or anticipated.

Most systems will respond to increased load with some degree of decreasing performance. A system's ability. to accept higher load is called scalability and modifying a system to handle a higher load is synonymous to performance tuning.

Systematic tuning follows these steps:

1. Assess the problem and establish numeric values that categorize acceptable behaviour.

2. Measure the performance of the system before modification.

3. Identify the part of the system that is critical for improving the performance.

4. Modify that part of the system to remove the bottleneck.

5. Measure the performance of the system after modification.

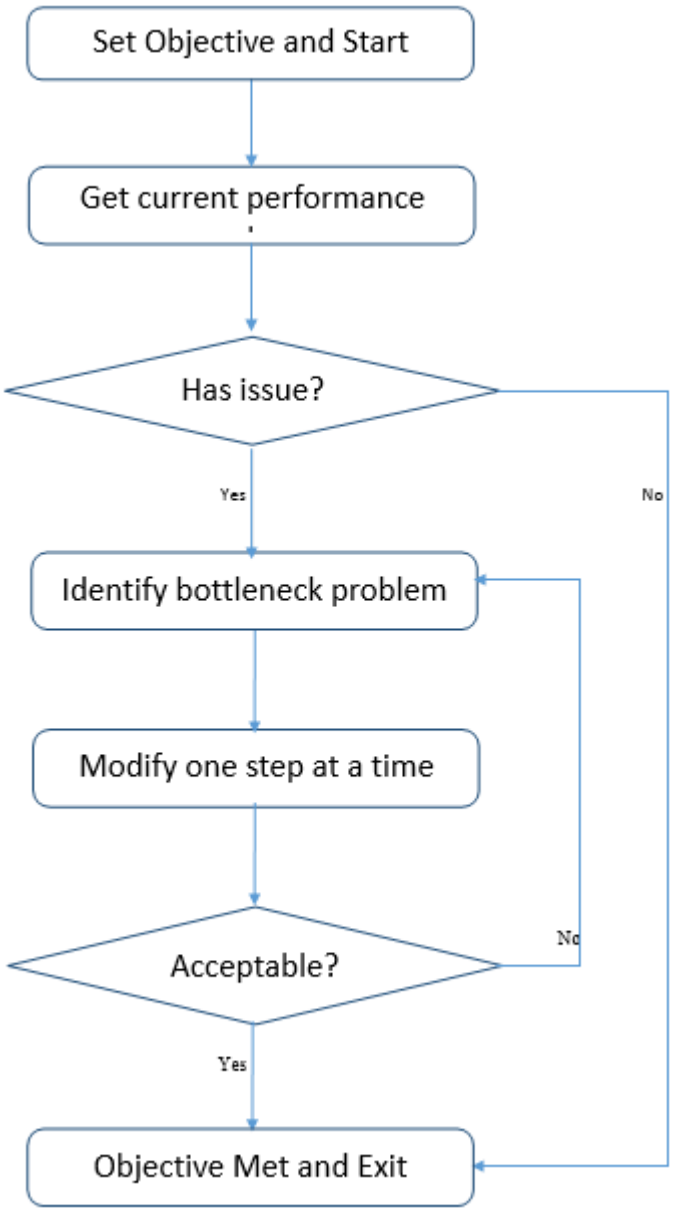


Fig: Performance tuning steps with a flow chart

MATRIX MULTIPLICATION..,

MATRIX

It's contain a set of value in which sequence of **row** and **column** an order manly.

Example :

MATRIX MULTIPLICATION

For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

matrix. The resulting matrix, known as
the matrix product, has the number of rows of the first and the number of columns of the second matrix.

A parallel algorithms for multiplying two n x n dense, **square matrices** A and B to yield the product matrix **C = A x B .**

Multiplying an **m×n** matrix **A** with m rows and n columns and an **n×l** matrix **B** with n rows and l columns produces an matrix C an **m×l** with m rows and l columns .

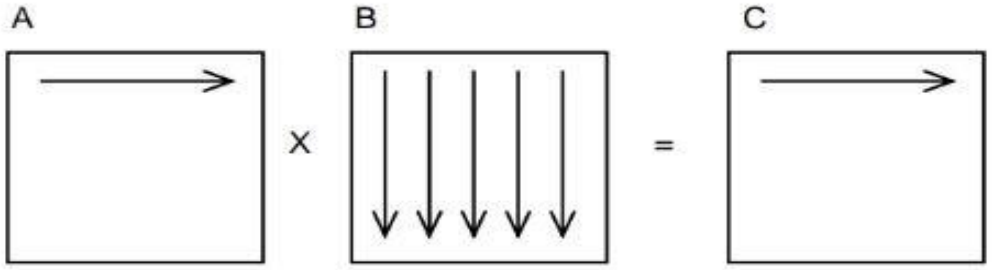
Each element of the matrix C is *calculated* according to the formula;

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l .$$

In case of square matrices, the size of which is n×n , the number of the executed operations is the order O(n3).

There are also *sequential matrix multiplication algorithms of smaller computational complexity.*

Figure of sequential matrix algorithm,



SEQUENTIAL ALGORITHM

// **Sequential matrix multiplication algorithm**

```
double MatrixA[Size][Size]; double
MatrixB[Size][Size]; double
MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++)
        {
        for (j=0; j<Size; j++)
        {
        MatrixC[i][j] = 0;
        for (k=0; k<Size; k++)
                {
                MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
                }
                }
        }
```

MATRIX MULTIPLICATION EXECUTIVE WAYS

Two of them are based on **block-striped data decomposition scheme.**

The other two methods are based on **checkerboard block scheme decomposition**. They are,

The Fox algorithm

The Cannon method.

Example for MATRIX MULTIPLICATION (3 X 3) :

```
#include<iostream>
int main()
{
        int
        a[10][10],b[10][10],mul[10][1
        0],r,c,i,j,k;
        cout<<"enter the number of
        row=";      cin>>r;
        cout<<"enter the number of
        column=";      cin>>c;
        cout<<"enter the first matrix
        element=\n";
        for(i=0;i<r;i++)
                {
                for(j=0;j<c;j++)
                {
```

```
                cin>>a[i][j];
                }
                }
        cout<<"enter the second matrix element=\n";
        for(i=0;i<r;i++)
                {
                for(j=0;j<c;j++)
                {
                cin>>b[i][j];
                }
                }
        cout<<"multiply of the matrix=\n";
        for(i=0;i<r;i++)
                {
                for(j=0;j<c;j++)
                {
                    Mul[i][j]=0;
                    for(k=0;k<c;k++)
                        {
                        mul[i][j]+=a[i][k]*b[k][j];
                        }
                }
                }
        //for printing result
        for(i=0;i<r;i++)
                {
                for(j=0;j<c;j++)
                {
                cout<<mul[i][j]<<" ";
                }
                cout<<"\n";
                }
        return 0;
}
```

OUTPUT :

```
enter the number of row = 3
enter the number of column = 3
enter the first matrix element=
2 3 4
6 5 3
7 8 7
```

enter the second matrix element=

7 3 4

8 5 3

7 6 2

Multiply of matrix=

56 45 25

103 61 45

162 103 66

Sorting

- Arrange elements of a list into certain order
- Make data become easier to access
- Speed up other operations such as searching and merging
- Many sorting algorithms with different time and space complexities

Parallel Sorting

Design methodology

- Based on an existing sequential sort algorithm
  - Try to utilize all resources available
  - Possible to turn a poor sequential algorithm into a reasonable parallel algorithm (bubble sort and parallel bubble sort)
- Completely new approach
  - New algorithm from scratch
  - Harder to develop
  - Sometimes yield better solution

Bubble Sort

- One of the straight-forward sorting methods

6 5 3 1 8 7 2 4

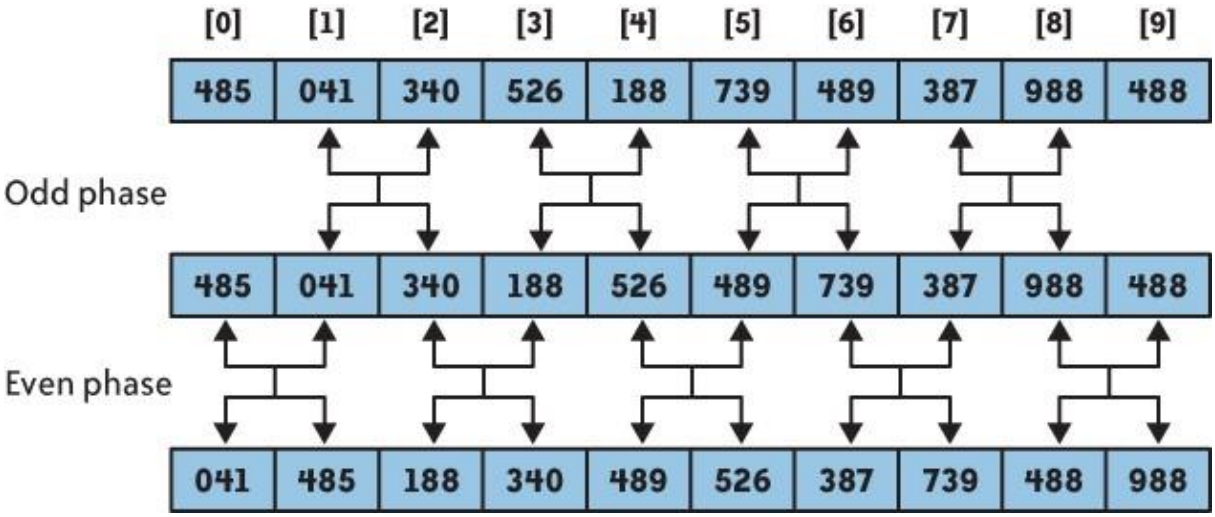
  - Cycles through the list
  - Compares consecutive elements and swaps them if necessary
  - Stops when no more out of order pair

- Slow & inefficient
- Average performance is $O(n^2)$

Parallel Bubble Sort

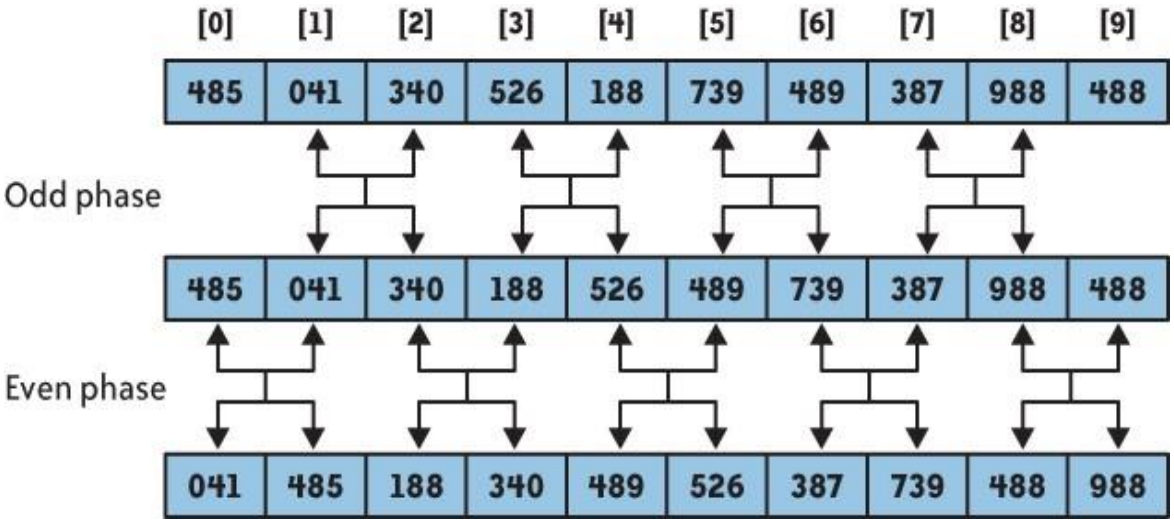
Odd-even Transposition Sort

- Compare all pairs in the list in parallel
- Alternate between odd and even phases



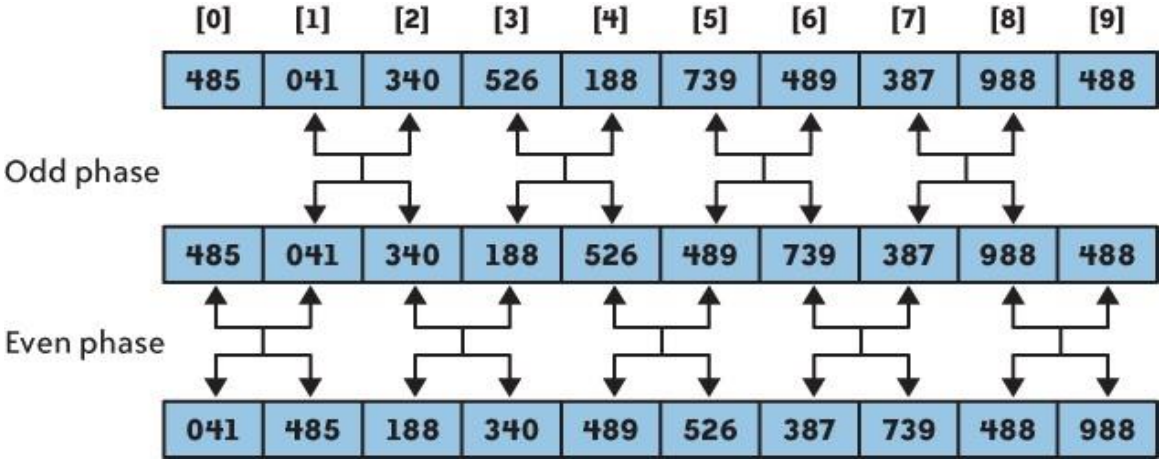
Parallel Bubble Sort

- When to stop?
- Shared flag, **sorted**, initialized to true at beginning of each iteration (2 phases), if any processor perform swap, **sorted** = false



Parallel Bubble Sort Complexity

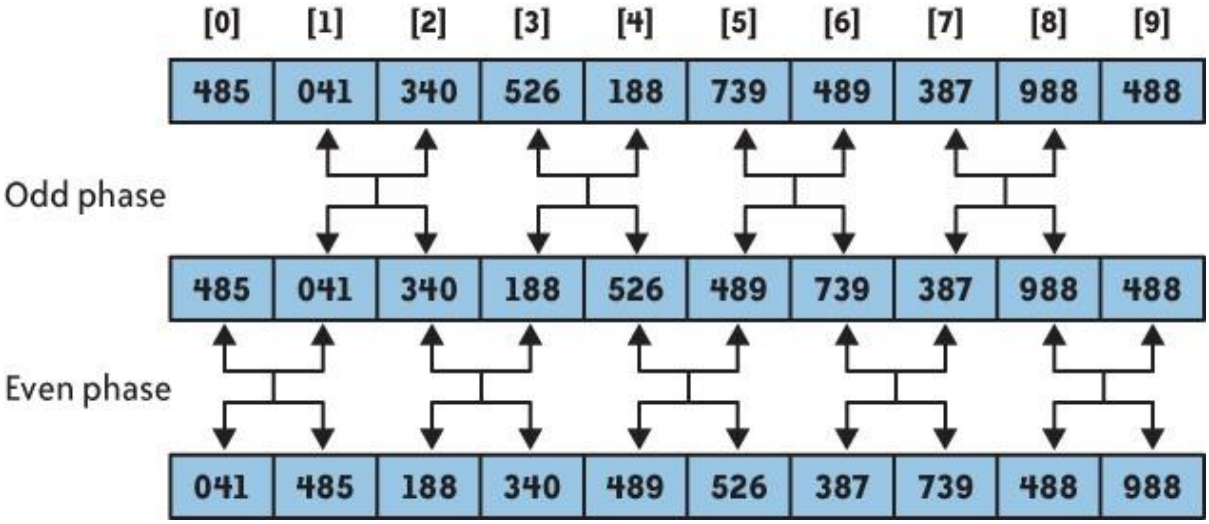
- Sequential bubble sort, $O(n^2)$
- Parallel bubble sort? (if we have unlimited # of processors)



Parallel Bubble Sort Complexity

- Sequential bubble sort, $O(n^2)$

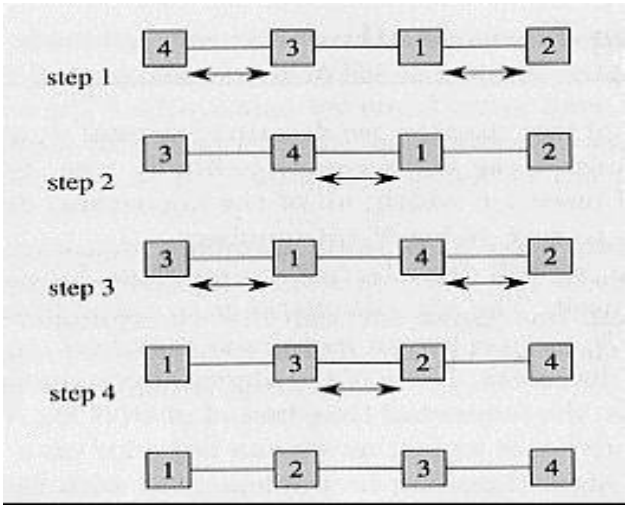
- Parallel bubble sort?
- Do n-1 comparisons for each iteration => O(n)
- Seq. Quicksort is O(nlogn)



Parallel Bubble Sort Example

- How many steps does it take to sort the following sequence from least to greatest using the Parallel Bubble Sort? How does the sequence look like after 2 cycles?
- 4,3,1,2

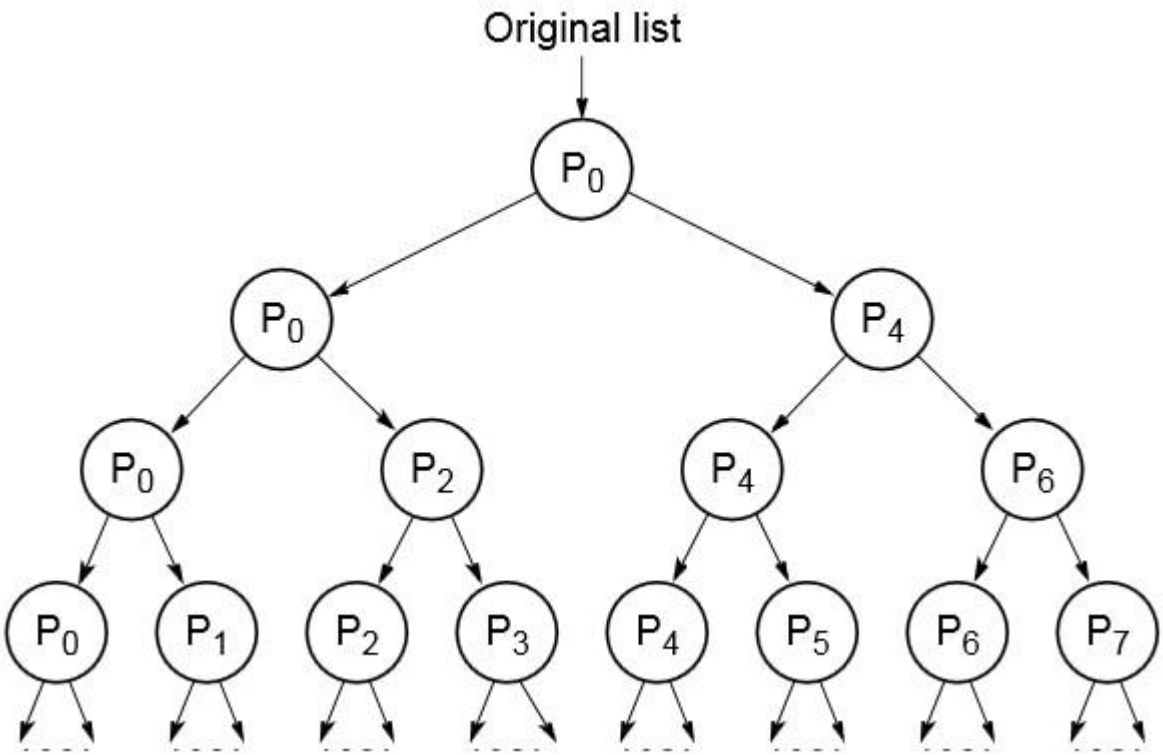
Parallel Bubble Sort Example



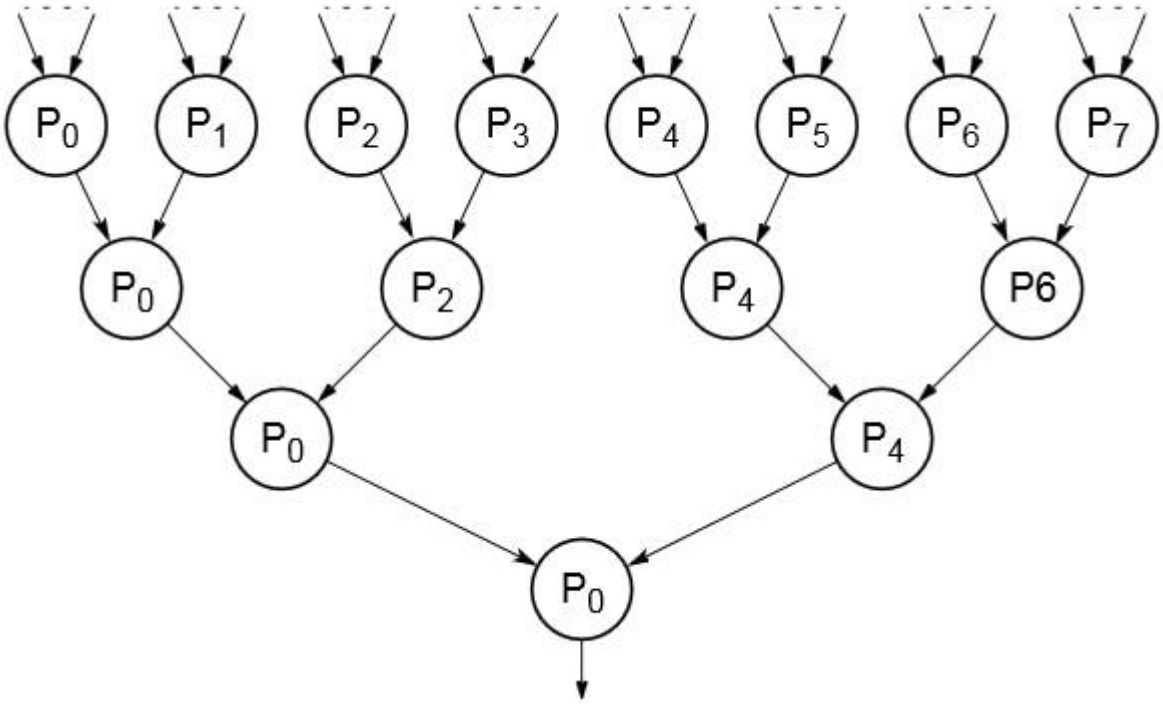
Divide and Conquer

- Dividing problem into sub-problems
- Division usually done through recursion
- Solutions from sub-problems then combined to give solution to the original problem

“Divide”

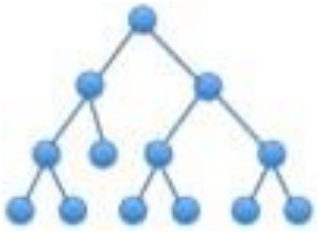


“Conquer”

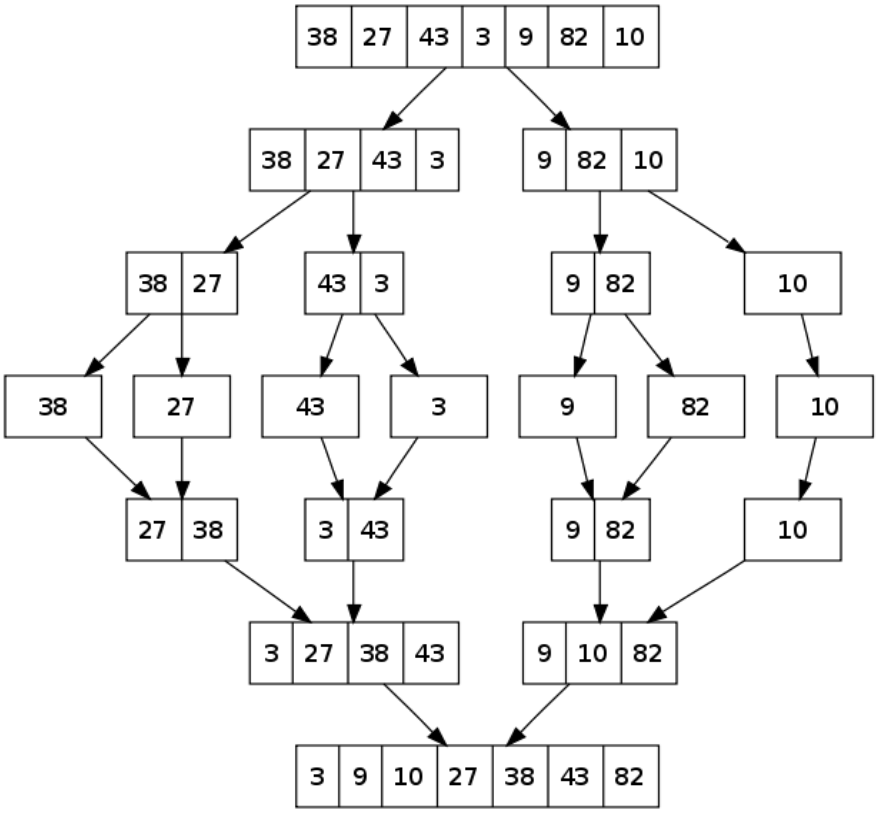


Merge Sort

- Collects sorted list onto one processor
- Merges elements as they come together
- Simple tree structure
- Parallelism is limited when near the root



Example



Merge Sort Complexity

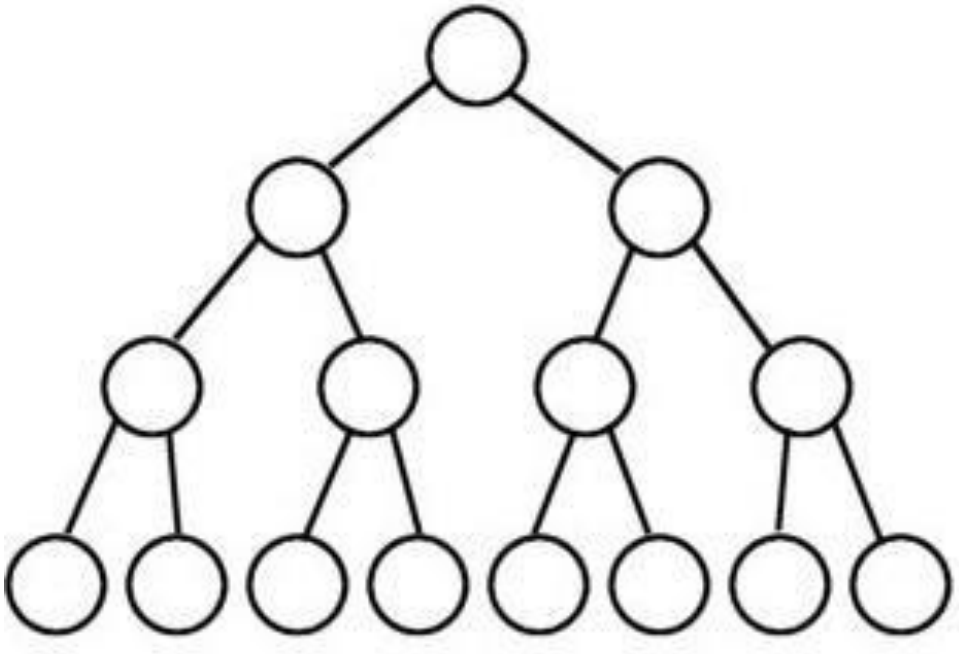
$$T(n) = \begin{cases} b & n = 1 \\ 2T\left(\frac{n}{2}\right) + bn & n > 1 \end{cases}$$

Solve the recurrence relation, (CS331)

T(n) = O(nlogn)

Parallel Merge Sort

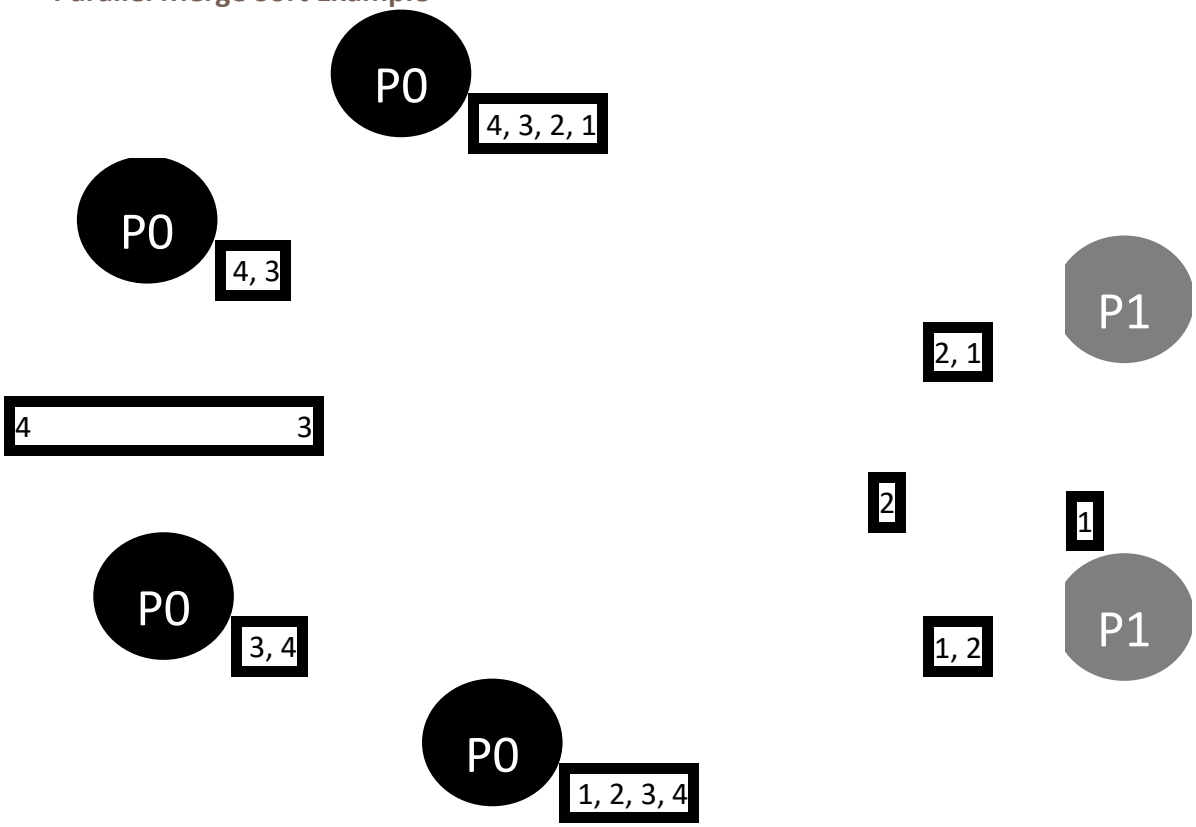
- Parallelize processing of sub-problems
- Max parallelization achived with one processor per node (at each layer/height)



Parallel Merge Sort Example

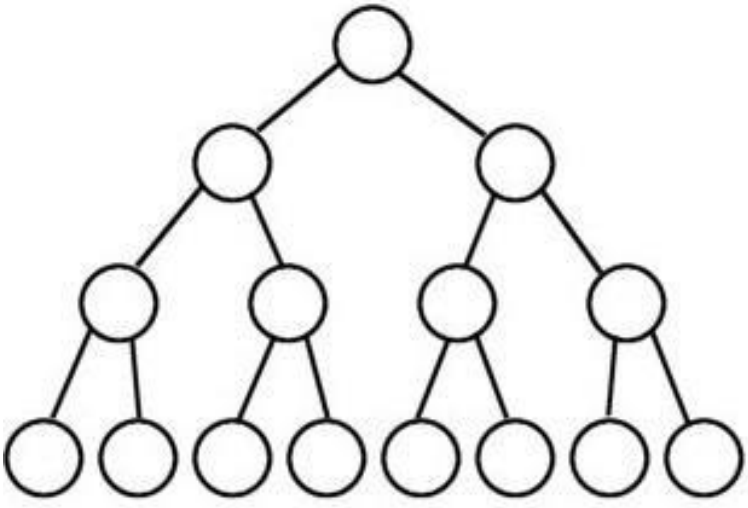
- Perform Merge Sort on the following list of elements. Given 2 processors, P0 & P1, which processor is reponsible for which comparison?
- 4,3,2,1

Parallel Merge Sort Example



Parallel Merge Sort Complexity

- Merge sort, O(nlogn)
- Easy way to remember complexity, n (elements) x logn (tree depth)
- If we have n processors, O(logn)



Parallel algorithm – searching

Searching is one of the fundamental operations in computer science. It is used in all applications where we need to find if an element is in the given list or not. In this chapter, we will discuss the following search algorithms

- Divide and Conquer
- Depth-First Search
- Breadth-First Search
- Best-First Search

Divide and Conquer

In divide and conquer approach, the problem is divided into several small subproblems. Then the sub-problems are solved recursively and combined to get the solution of the original problem.

The divide and conquer approach involves the following steps at each level −

- **Divide** − The original problem is divided into sub-problems.
- **Conquer** − The sub-problems are solved recursively.
- **Combine** − The solutions of the sub-problems are combined to get the solution of the original problem.

Depth-First Search

Depth-First Search (or DFS) is an algorithm for searching a tree or an undirected graph data structure. Here, the concept is to start from the starting node known as the **root** and traverse as far as possible in the same branch. If we get a node with no successor node, we return and continue with the vertex, which is yet to be visited.

Steps of depth-first search

- Consider a node (root) that is not visited previously and mark it visited.
- Visit the first adjacent successor node and mark it visited.
- If all the successors nodes of the considered node are already visited or it doesn't have any more successor node, return to its parent node.

Breadth-First Search

Breadth-First Search (or BFS) is an algorithm for searching a tree or an undirected graph data structure. Here, we start with a node and then visit all the adjacent nodes in the same level and then move to the adjacent successor node in the next level. This is also known as **level-by-level search**.

Steps of Breadth-First Search

- Start with the root node, mark it visited.
- As the root node has no node in the same level, go to the next level.
- Visit all adjacent nodes and mark them visited.
- Go to the next level and visit all the unvisited adjacent nodes.
- Continue this process until all the nodes are visited.

Best-First Search

Best-First Search is an algorithm that traverses a graph to reach a target in the shortest possible path. Unlike BFS and DFS, Best-First Search follows an evaluation function to determine which node is the most appropriate to traverse next.

Steps of Best-First Search

- Start with the root node, mark it visited.
- Find the next appropriate node and mark it visited.
- Go to the next level and find the appropriate node and mark it visited.
- Continue this process until the target is reached.

Minimum Spanning Trees

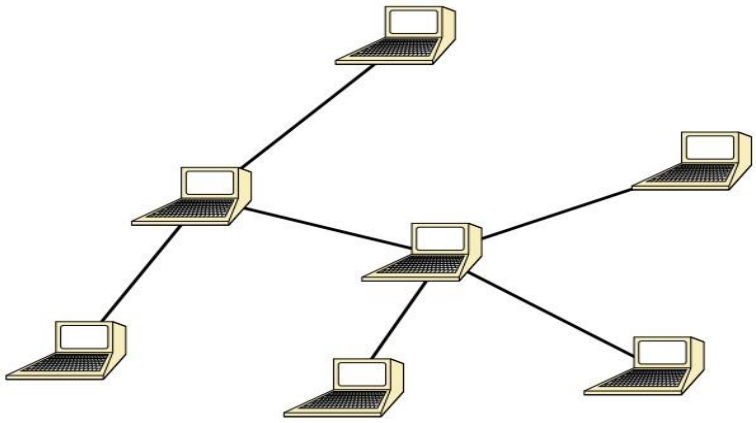
- Find a minimum-cost set of edges that connect all vertices of a graph

- Applications

– Connect “nodes” with a minimum of “wire”

- Networking

- Circuit design



Minimum Spanning Trees

- Find a minimum-cost set of edges that connect all vertices of a graph

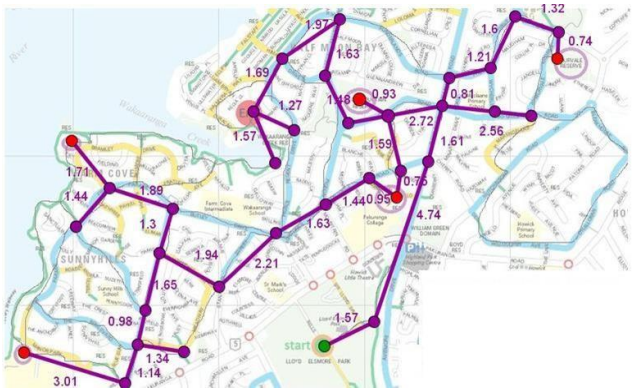
- Applications
  – Collect nearby nodes

- Clustering, taxonomy construction

Minimum Spanning Trees

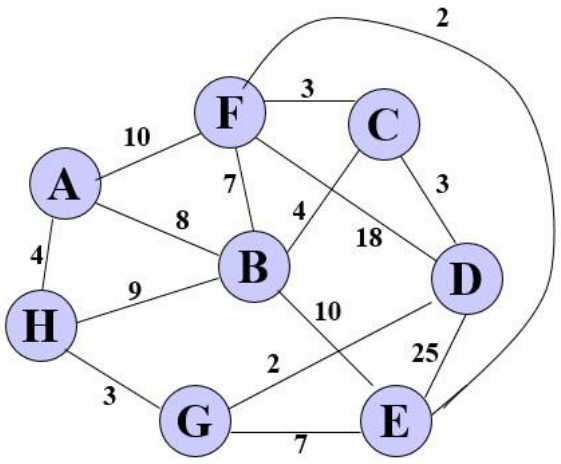
- Find a minimum-cost set of edges that connect all vertices of a graph

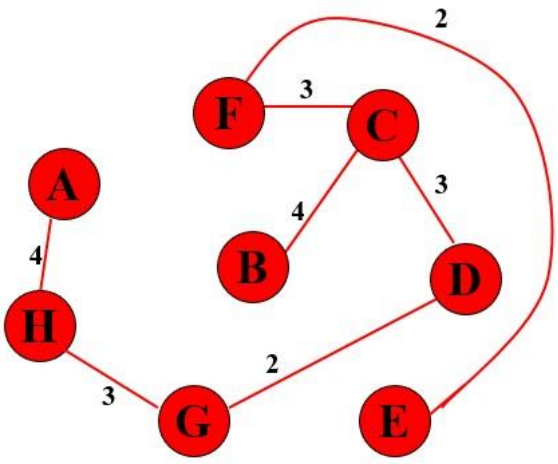
- Applications
  – Approximating graphs

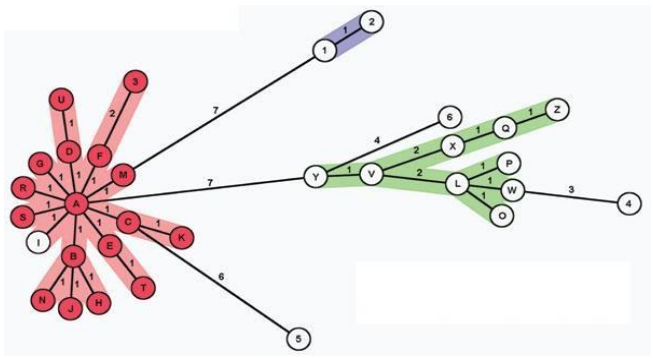


- A tree is an acyclic, undirected, connected graph
- A spanning tree of a graph is a tree containing all vertices from the graph
- A minimum spanning tree is a spanning tree, where the sum of the weights on the tree's edges are minimal

Minimum Spanning Trees







A minimum spanning tree is a spanning tree, where the sum of the weights on the tree's edges are minimal

- Minimum spanning tree may be not unique (can be more than one)

Minimum Spanning Trees

- Problem formulation

— Given an undirected, weighted graph $G = (V, E)$ with weights for each edge $(u, v) \in E$

— Find an acyclic subset $T \subseteq E$ that connects all of the vertices V and minimizes the total weight:

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

(V, T)

— The minimum spanning tree is

- Minimum spanning tree may be not unique (can be more than one)

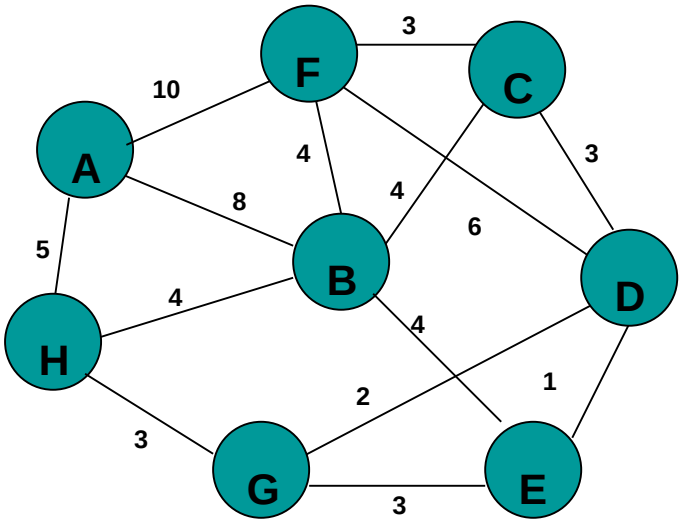
Minimum Spanning Trees

- Both Kruskal's and Prim's Algorithms work with undirected graphs
- Both work with weighted and unweighted graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions

Minimum Spanning Trees

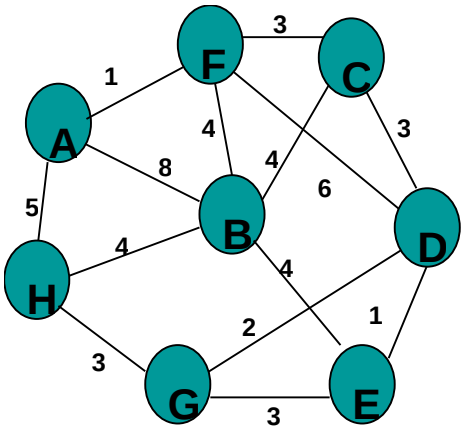
- Solution 1: Kruskal's algorithm
  - Work with edges – Two steps:
- Sort edges by increasing edge weight
- Select the first $|V| - 1$ edges that do not generate a cycle

- Walk through:



Minimum Spanning Trees

- Solution 1: Kruskal's algorithm



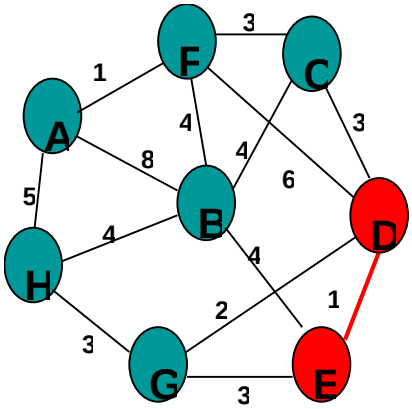
Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Minimum Spanning Trees

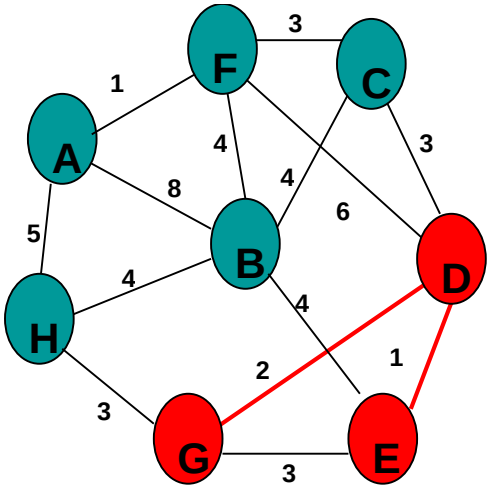
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	?
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

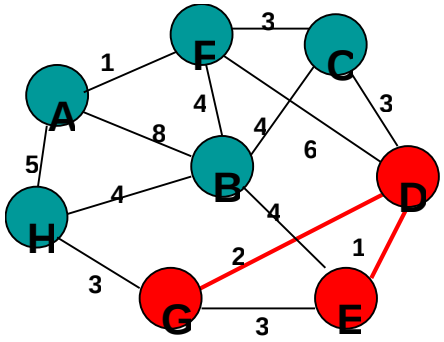
Minimum Spanning Trees



edge	d_v	
(D,E)	1	?
(D,G)	2	?
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

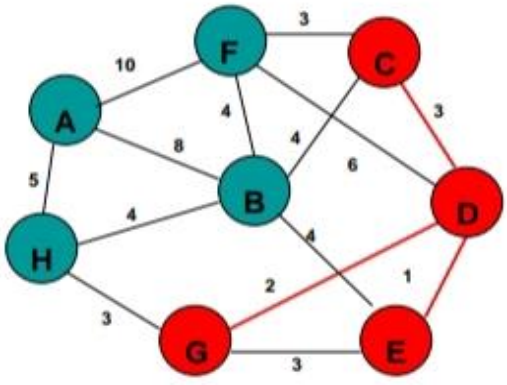
Minimum Spanning Trees



edge	d_v	
(D,E)	1	?
(D,G)	2	?
(E,G)	3	?
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

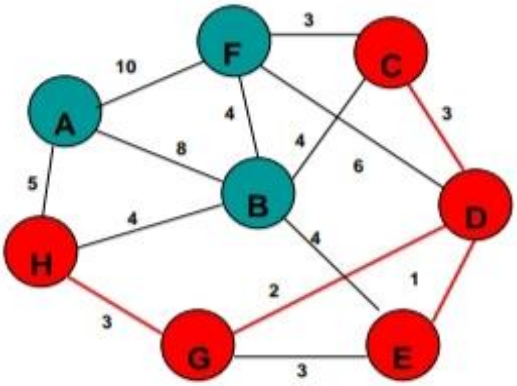
Accepting edge (E,G) would create a cycle



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

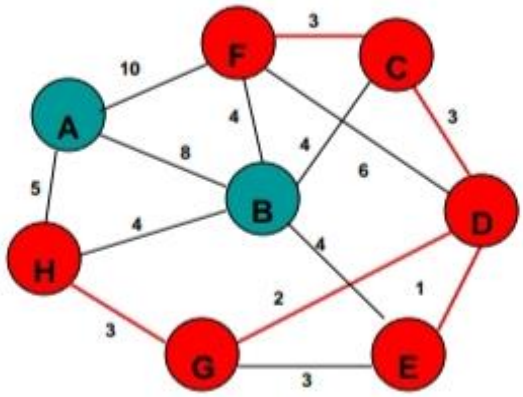
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

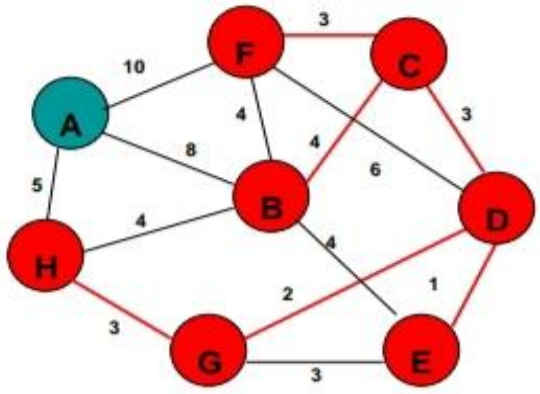
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

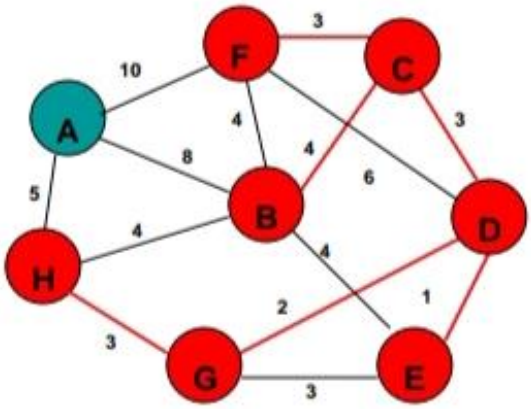
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

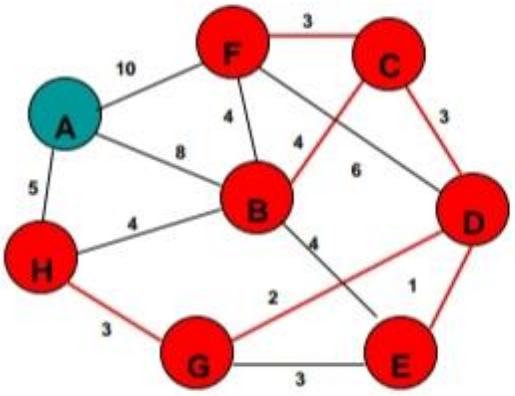
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

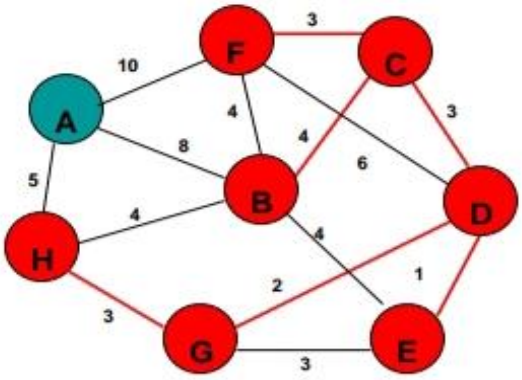
edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

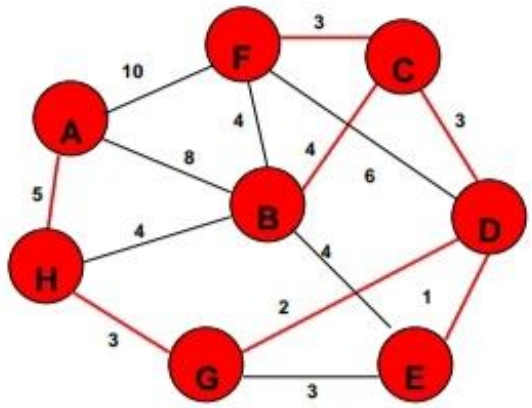
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

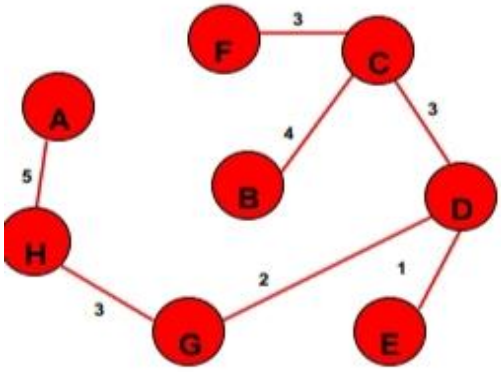
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

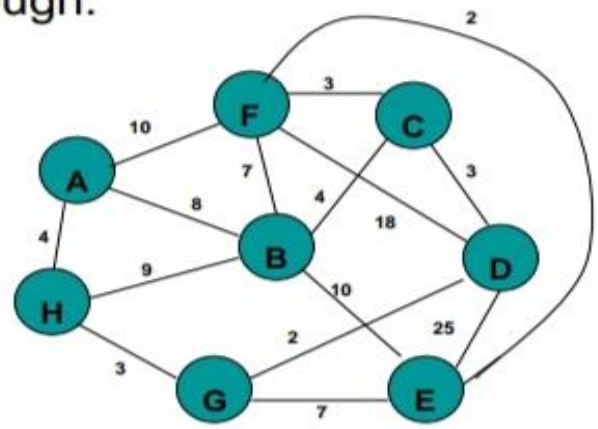
edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

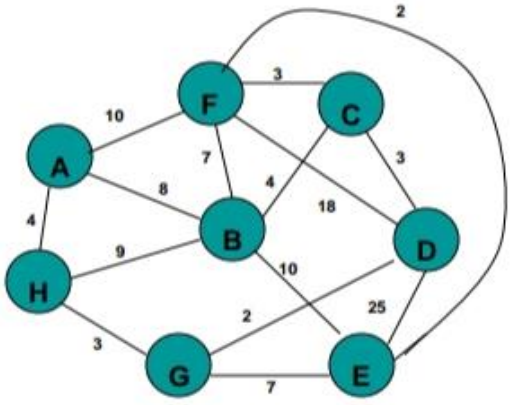
Total Cost = *21*

- Solution 2: Prim's algorithm
  - Work with nodes (instead of edges)
  - Two steps
    - Select node with minimum distance
    - Update distances of adjacent, unselected nodes
  - Walk through:



Minimum Spanning Trees

- Solution 2: Prim's algorithm



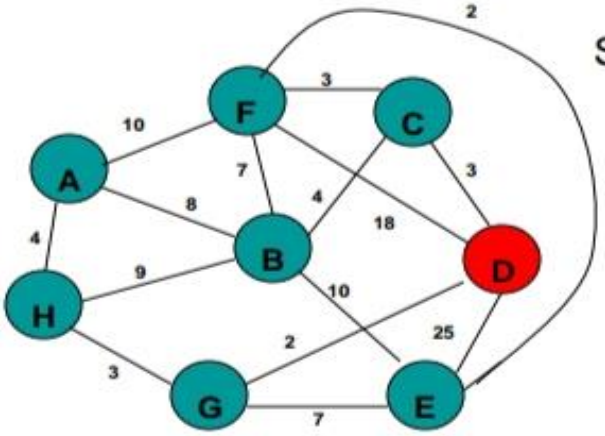
Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–

K : whether in the tree

d_v : distance to the tree

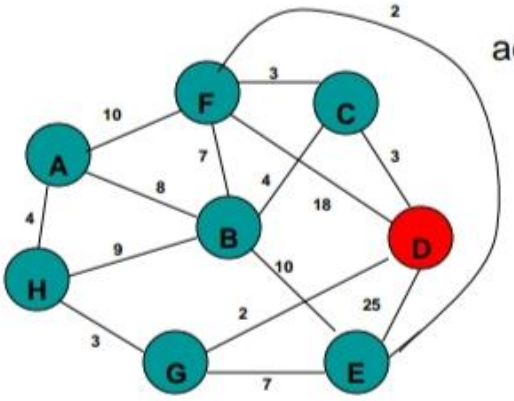
p_v : closest node that is in the tree



Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			

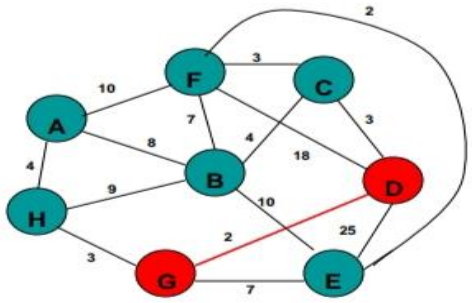
- Solution 2: Prim's algorithm



Update distances of adjacent, unselected nodes

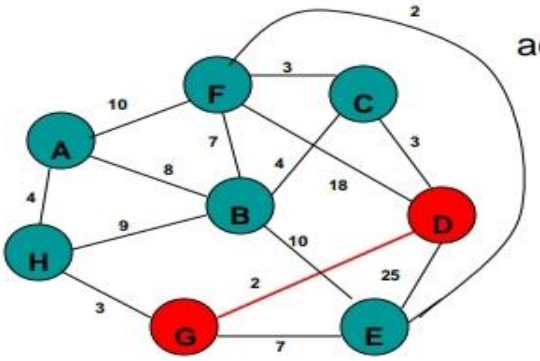
	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			

- Solution 2: Prim's algorithm



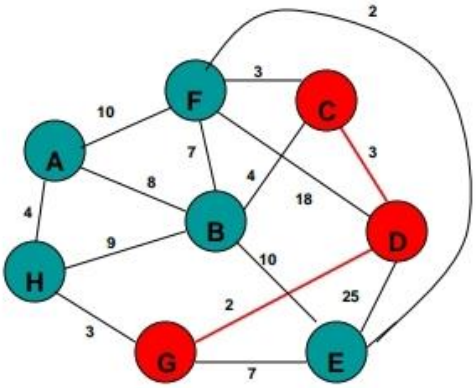
Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



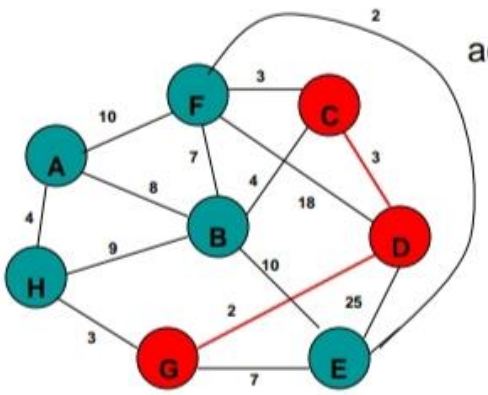
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



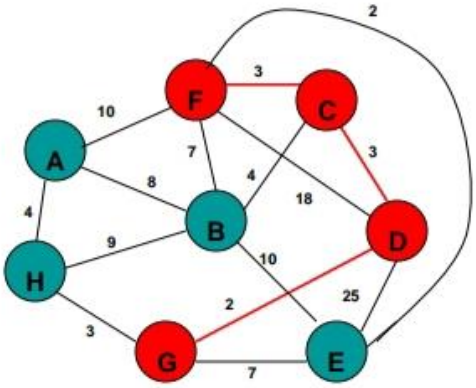
Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



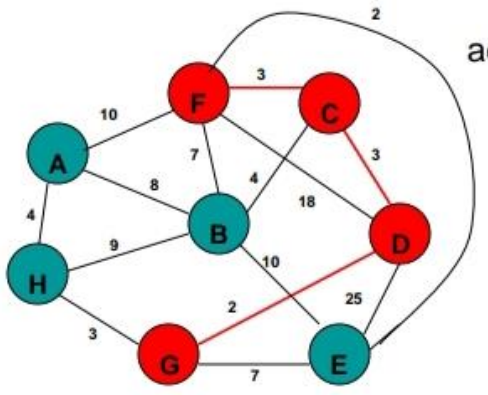
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



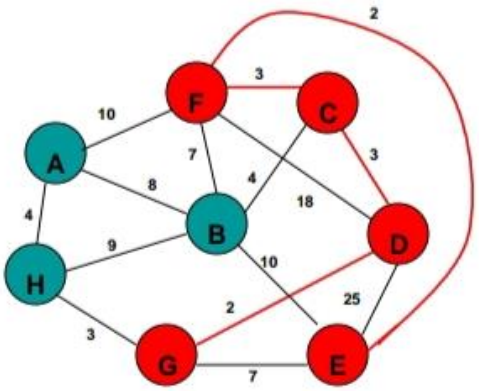
Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



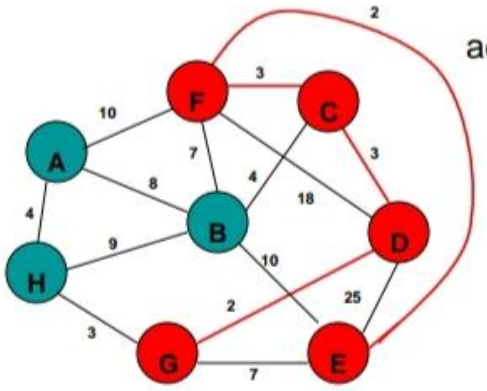
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

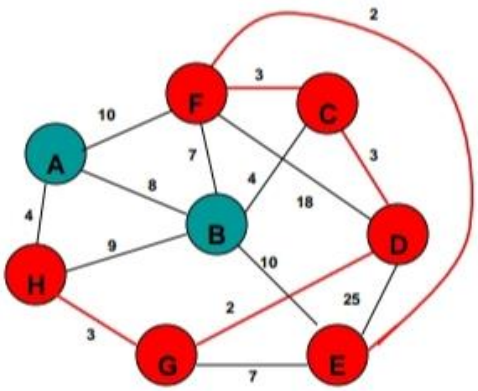
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

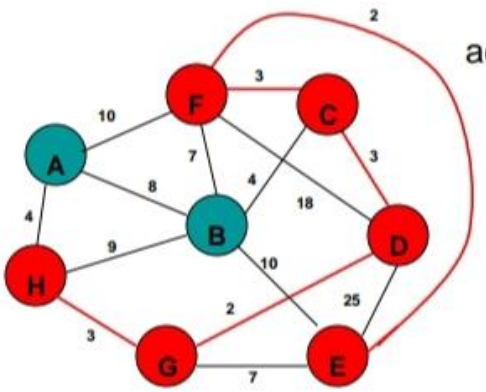
	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



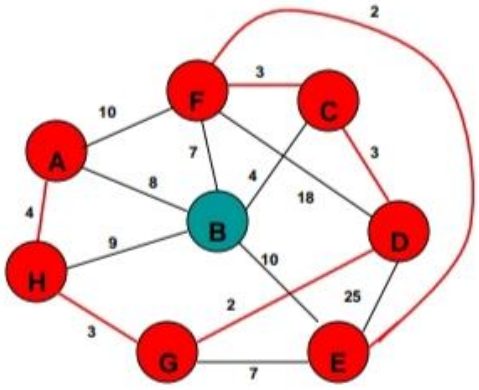
Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



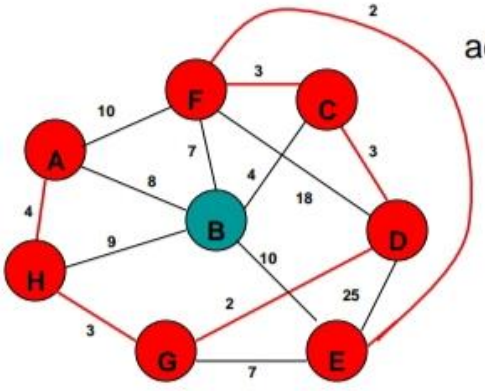
Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

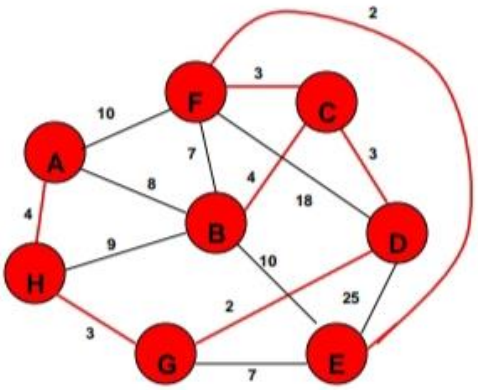
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

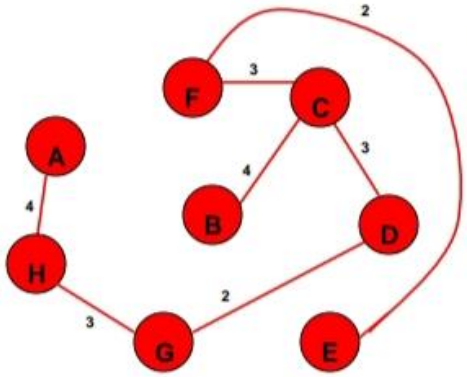
	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = **21**

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

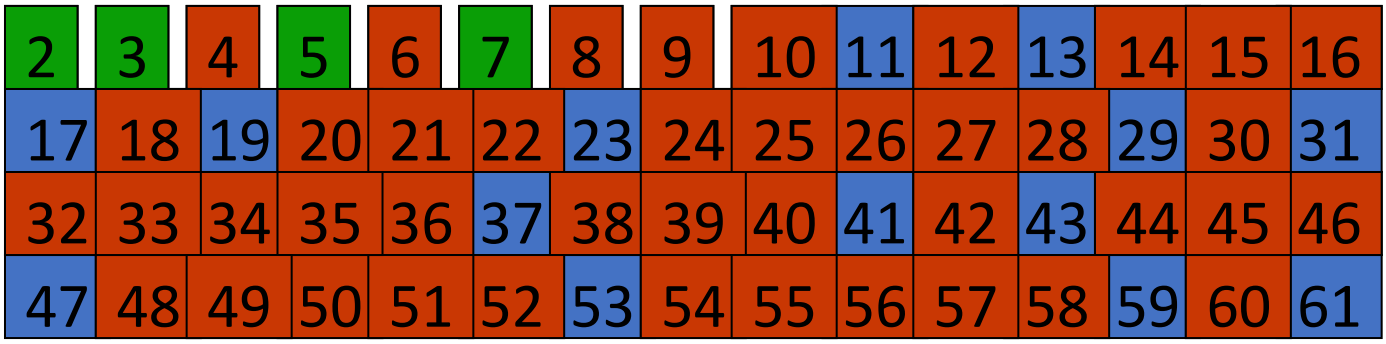
Done

Minimum Spanning Trees

- Runtime
  - When using binary heaps, the runtime of the Kruskal's algorithm is $O(E \lg V)$
  - When using binary heaps, the runtime of the Prim's algorithm is $O(E \lg V)$
    When using the Fibonacci heaps, the runtime of the Prim's algorithm becomes: $O(E + V \lg V)$
  - So, when an undirected graph is dense (i.e., $|V|$ is much small than $|E|$), the Prim's algorithm is more efficient

Prime numbers.

Sequential Algorithm



- The algorithm is not practical for large values of n.
- Algorithm complexity is $\Theta(n \ln \ln n)$
- The number of digits of n is logarithmic to size of n
- Finding prime numbers is important for encryption

purposes.

- A modified form of sieve is an important research

tool in number theory.

Data Structure Used For Sequential Algorithm

- Assume a Boolean array of n elements
- Array indices are 0 through n-2 and they represent the numbers 2, 3, ..., n.
- The boolean value at index i represents whether of not the number i+2 is marked.
    - Indices that are marked represent composite numbers (i.e., not prime)
- Initially, all numbers are unmarked

One Method to Parallelize

- Because the focus of the algorithm is the marking of elements in an array, domain decomposition makes sense.
- Domain decomposition
- Divide data into n-1 pieces
- Associate computational steps with data

- One primitive task per array element
- These will be agglomerated into larger groups of elements.

Parallelizing Algorithm Step 3(a)

- Recall Step 3(a):

3 a) Mark all multiples of k between k^2 and n

- The following straightforward modification allows this to be computed in parallel:

for all j where $k^2 \leq j \leq n$ do

if j mod k = 0 then

mark j (i.e. it is not a prime)

endif

endfor

- Each j above represents a primitive task
- Recall Step 3(a):

3 b) Find smallest unmarked number > k

- Parallelizing requires two steps:
  - Min-reduction (to find smallest unmarked number > k)
  - Broadcast (to get result to all tasks)
- Plus- remember these are in a repeat-until loop which loops until $k^2 > n$.

Method

- Let $r = n \bmod p$
- If $r = 0$, all blocks have same size
- Else
  - First r blocks have size $\lceil n/p \rceil$
  - Remaining $p-r$ blocks have size $\lfloor n/p \rfloor$
- Example: p = 8 and n = 45

  Observe that r = 45 mod 8 = 5

  So first 5 blocks have size $\lceil 45/8 \rceil = 6$ and

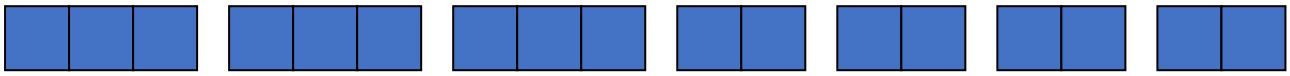
  the p-r = 8-5 = 3 others have size $\lfloor 45/8 \rfloor = 5$.

We've divided 45 items into 8 blocks as follows:

5 blocks of 6 items, then 3 blocks of 5 items

Examples

17 elements divided among 7 processes

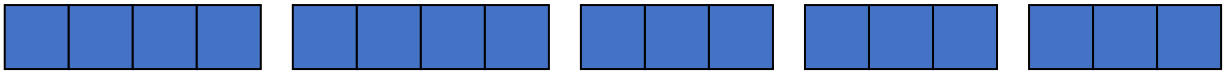


17 elements divided among 5 processes

17 elements divided among 3 processes

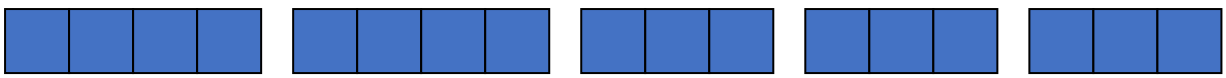
- Let $r = n \bmod p$
- The first element controlled by process i is
- Example: The first element controlled by process 1 is 1*3 + min(1,2) = 4 in below example:

17 elements divided among 5 processes



- Let $r = n \bmod p$
- Last element controlled by process i
- Note this is just the element immediately before the first element controlled by process i+ 1.
- Example: The last element controlled by process 2 is (2+1)*3 + min(2+1,2) -1 = 3*3+2-1 = 10.

17 elements divided among 5 processes



- Let $r = n \bmod p$
- Process controlling element j

Example: The process controlling element j = 6 is

$\min(\lfloor 6/4 \rfloor, \lfloor 4/3 \rfloor) = \min(1,1) = 1$.

17 elements divided among 5 processes

- Although deriving the expressions could be a hassle, the expressions themselves are not too complicated to compute.
- The only worrisome one is the last one, where given an element index, we need to compute the controlling process.
- This action must be done repeatedly and while $\lfloor n/p \rfloor$ can be precalculated, the two divisions can't because they involve j.
- So, we'll try for another block data decomposition.

---------- X END X ----------