UNIT 5

Task partitioning (TP)

TP is the division of a task into 2 or more sub-tasks e.g. nectar collection & storage versus collection, storage
Division of Labour = Workers / Tasks
Task Partitioning = Task / Workers
Honey bees and stingless bees have task partitioning as above. Bumble bees so not. A nectar forager bumble bee also stores the nectar.

Task partitioning is the division of a task into two or more sub-tasks. If a load of forage is passed from one worker to another this is task partitioning. In honey bees and stingless bees, a nectar forager transfers her nectar to one or more nectar receiver bees in the nest. This is similar to a "bucket brigade" or assembly line. All known examples of TP involve the handling of material. The two sub-tasks are connected by the flow of material between them.

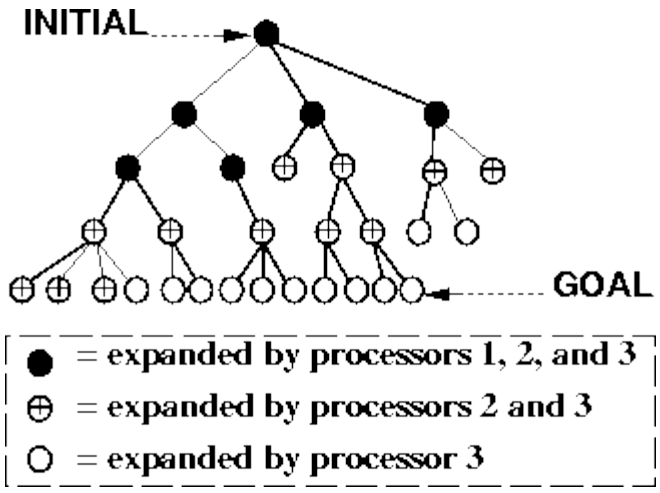
Data sharing

The ability to share the same data resource with multiple applications or users. It implies that the data are stored in one or more servers in the network and that there is some software locking mechanism that prevents the same set of data from being changed by two people at the same time. Data sharing is a primary feature of a database management system (DBMS).

Task Distribution

A search algorithm implemented on a parallel system requires a balanced division of work between contributing processors to reduce idle time and minimize redundant or wasted effort. One method of dividing up the work is with a parallel window search (PWS), introduced by Powley and Korf. Using PWS, each processor is given a copy of the entire search tree and a unique cost threshold. The processors search the same tree to different thresholds simultaneously. If a processor completes an iteration without finding a solution, it is given a new unique threshold (deeper than any threshold yet searched) and begins a new search pass with the new threshold. When an optimal solution is desired, processors that find a goal node must remain idle until all processors with lower cost thresholds have completed their current iteration. A typical division of work using PWS is illustrated in Figure 1.

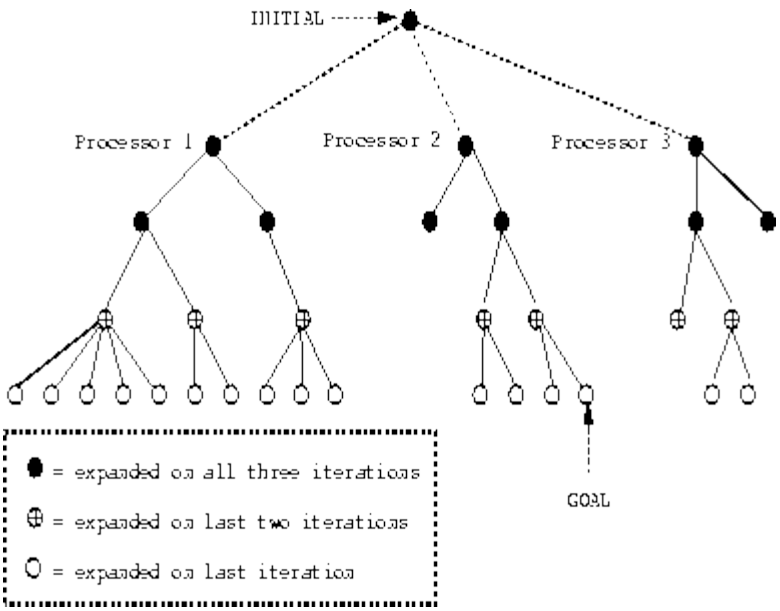
Figure 1: Division of work in parallel window search



One advantage of parallel window search is that the redundant search inherent in IDA* (Iterative Deepening A search) is not performed serially. During each non-initial iteration of IDA*, all of the nodes expanded in the previous iteration are expanded again. Using multiple processors, this redundant work is performed concurrently. A second advantage of parallel window search is the improved time in finding a first solution. If a search space holds many goal nodes, IDA* may find a deep solution much more quickly than an optimal solution. Parallel window search can take advantage of this type of search space. Processors that are searching beyond the optimal threshold may find a solution down the first branch they explore, and can return that solution long before other processors finish their iteration. This may result in superlinear speedup because the serial algorithm conservatively increments the cost threshold and does not look beyond the current threshold.

On the other hand, parallel window search can face a decline in efficiency when the number of processors is significantly greater than the number of iterations required to find an optimal (or a first) solution, causing all remaining processors to sit idle. This situation will occur when many processors are available, yet few iterations are required because the heuristic estimate is fairly accurate.

Figure 2: Division of work in distributed tree search

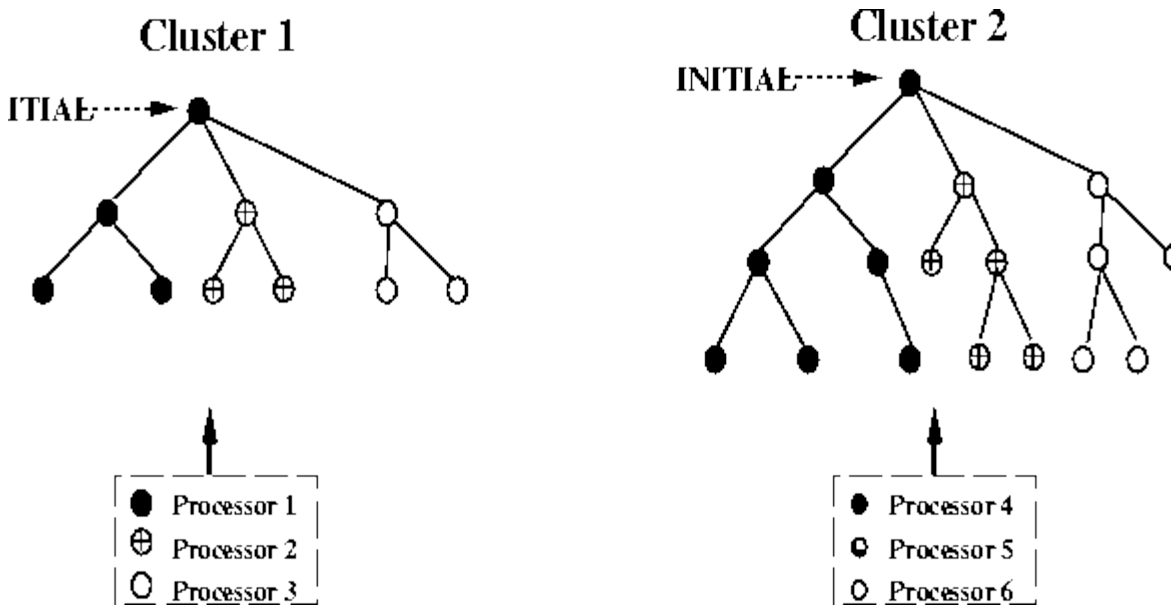


An alternative parallel search approach relies on distributing the tree among the processors. With this approach, the root node of the search space is given to the first processor and other processors are assigned subtrees of that root node as they request work. As an alternative, the distributed tree search algorithm (DTS) employs breadth-first expansion until there are at least as many expanded leaf nodes as available processors. Processors receive unique nodes from the expanding process and are responsible for the entire subtree rooted at the received node. Communication-free versions of this distribution scheme have also been reported. In all of these tree distribution approaches, the processors perform IDA* on their unique subtrees simultaneously. All processors search to the same threshold. After all processors have finished a single iteration, they begin a new search pass through the same set of subtrees using a larger threshold. A sample distribution of the search space is shown in Figure 2.

One advantage of this distribution scheme is that no processor is performing wasted work beyond the goal depth. Because the algorithm searches the space completely to one threshold before starting the search to a new threshold, none of the processors is ever searching at a level beyond the level of the optimal solution. It is possible, however, for DTS to perform wasted work at the goal depth. For example, in Figure 2 processor 3 searches nodes at the goal level that would not be searched in a serial search algorithm moving left-to-right through the tree.

A disadvantage of this approach is the fact that processors are often idle. To ensure optimality, a processor that quickly finishes one iteration must wait for all other processors to finish before starting the next iteration. This idle time can make the system very inefficient and reduce the performance of the search application. The efficiency of this approach can be improved by performing load balancing between neighboring processors working on the same iteration.

Figure 3: Space searched by two clusters, each with 3 processors



These described approaches offer unique benefits. Parallel window search is effective when many iterations of IDA* are required, when the tree is so imbalanced that DTS will require excessive load balancing, or when a deep, non-optimal solution is acceptable. On the other hand, dividing the search space among processors can be more effective when the branching factor is very large and the number of IDA* iterations is relatively small. A compromise between these approaches divides the set of processors into *clusters*. Each cluster is given a unique cost threshold, and the search space is divided between processors within each cluster, as shown in Figure 3. Setting the number of clusters to one simulates distributed tree search, and setting the number of clusters to the number of available processors simulates parallel window search.

Shared memory

Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs.

Race

A race condition is a programming fault producing undetermined program state and behavior due to un-synchronized parallel program executions. Race condition is the most worried programming fault by experienced programmers in parallel programming space. However there are many subtle aspects of race condition issues. A race condition problem is often caused by common data accessing, but it can also occur in a sequence of operations which require a protection such as atomic transaction to ensure the overall state integrity. Not every data race case is a programming bug. There is a compromised aspect of allowing race condition in a parallel program for performance reason.

Dependence

A **data dependency** is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

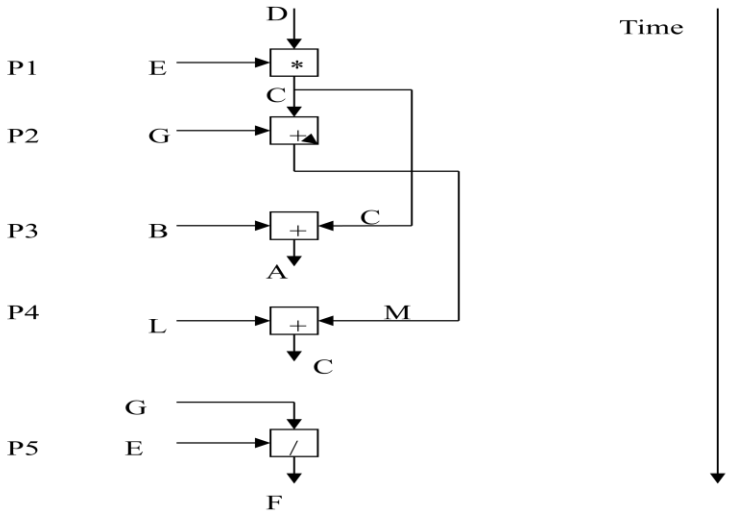
There are three types of dependencies: data, name, and control.

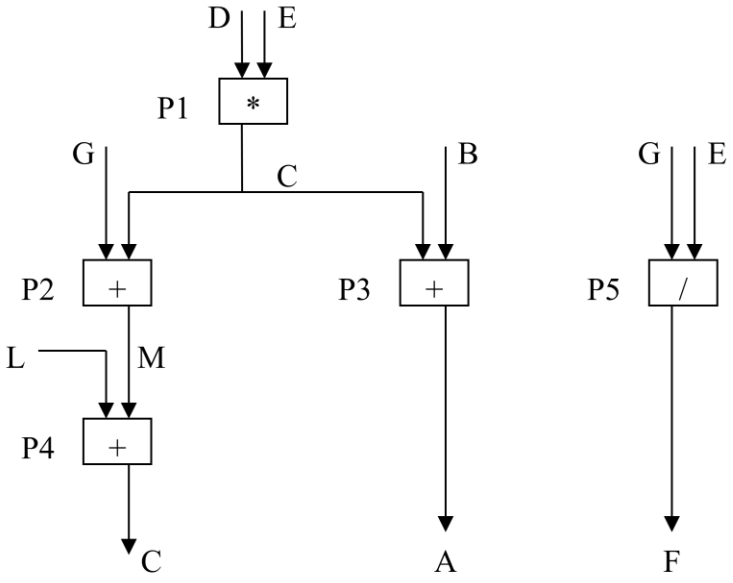
Bernstein's conditions for parallelism

- Define:
  - I_i as the input set of a process P_i
  - O_i as the output set of a process P_i
  - P_1 and P_2 can execute in parallel (denoted as $P_1 \parallel P_2$) under the condition:
  - $I_1 \cap O_2 = 0$
  - $I_2 \cap O_1 = 0$
  - $O_1 \cap O_2 = 0$
  - Note that $I_1 \cap I_2 <> 0$ does not prevent parallelism
- Input set: also called read set or domain of a process
- Output set: also called write set or range of a process
- A set of processes can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis; that is, $P_1 \| P_2 \| \ldots \| P_K$ if and only if $P_i \| P_j$ for all i<>j
- The parallelism relation is commutative: Pi || Pj implies that Pj || Pi
- The relation is not transitive: Pi || Pj and Pj || Pk do not necessarily mean Pi || Pk
- Associativity: Pi || Pj || Pk implies that (Pi || Pj) || Pk = Pi || (Pj || Pk)
- For n processes, there are 3n(n-1)/2 conditions; violation of any of them prohibits parallelism collectively or partially
- Statements or processes which depend on run-time conditions are not transformed to parallelism. (IF or conditional branches)
- The analysis of dependences can be conducted at code, subroutine, process, task, and program levels; higher-level dependence can be inferred from that of subordinate levels

Example of parallelism using Bernstein's conditions

- P1: C = D * E
- P2: M = G + C
- P3: A = B + G
- P4: C = L + M
- P5: F = G / E
- Assume no pipeline is used, five steps are needed in sequential execution





- There are 10 pairs of statements to check against Bernstein's conditions
- Only P2 || P3 || P5 is possible because P2 || P3, P3 || P5 and P2 || P5 are all possible
- If two adders are available simultaneously, the parallel execution requires only three steps

Progress – lock and wait freedom

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread; for some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress, and wait-free if there is also guaranteed per-thread progress.

Wait-freedom is the strongest non-blocking guarantee of progress, combining guaranteed system-wide throughput with starvation-freedom. An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. This property is critical for real-time systems and is always nice to have as long as the performance cost is not too high. It was shown in the 1980s that all algorithms can be implemented wait-free, and many transformations from serial code, called *universal constructions*, have been demonstrated. However, the resulting performance does not in general match even naïve blocking designs. Several papers have since improved the performance of universal constructions, but still, their performance is far below blocking designs.

Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress (for some sensible definition of progress). All wait-free algorithms are lock-free. An algorithm is lock free if infinitely often operation from some processors will succeed in finite number of steps. For instance, if N number of processors are trying to make an operation, some processes in N number of processes will succeed to finish the operation in finite number of steps and other might fail and retry on failure. The difference between wait-free and lock-free is, wait-free operations by every process is guaranteed to succeeds in finite number of steps regardless of other processors.

In general, a lock-free algorithm can run in four phases: completing one's own operation, assisting an obstructing operation, aborting an obstructing operation, and waiting. Completing one's own operation is complicated by the possibility of concurrent assistance and abortion, but is invariably the fastest path to completion. The decision about when to assist, abort or wait when an obstruction is met is the responsibility of a *contention manager*. This may be very simple (assist higher priority operations, abort lower priority ones), or may be more optimized to achieve better throughput, or lower the latency of prioritized operations. Correct concurrent assistance is typically the most complex part of a lock-free algorithm, and often very costly to execute: not only does the assisting thread slow down, but thanks to the mechanics of shared memory, the thread being assisted will be slowed, too, if it is still running.

Dynamic Vs Static Scheduling

- Dynamic (On-line) Scheduling
  - Only considers

        - Actual requests
        - Execution time parameters
    - Costly to find a schedule
- Static Scheduling (Off-line)
    - Complete knowledge
        - Maximum execution time
        - Precedence constraints
        - Mutual exclusion constraints
        - Deadlines

Optimal Dynamic Scheduling

- Consider a dynamic scheduler with full past knowledge only
    - Exact schedulability is impossible
    - New definition of optimal dynamic scheduler
        - Optimal if it can find a schedule whenever a clairvoyant scheduler can find a schedule
- Dynamic scheduling algorithm
    - Determines task after occurrence of a significant event

Based on current task requests

Dynamic Scheduling In Distributed Systems

- Hard to guarantee deadlines in single processor systems
- Even harder in distributed systems or multiprocessor systems due to communication
- Applications required to tolerate transient faults like message losses as well as detect permanent faults
- Positive Acknowldgement or Retransmission (PAR)
    - Large temporal uncertainty between shortest and longest execution time
    - Worse case – assume longest time.

        - Poor responsiveness of system
- Masking Protocols
    - Send message k + 1 in case the tolerance of k is required.

No temporal problem but can't detect permanent faults due to unidirectional communication

Static Scheduling

- Static schedules guarantees all deadlines, based on known resources, precedence, and synchronization requirements, is calculated off-line
- Strong regularity assumptions
- Known times when external events will be serviced
- System design
    - Maximum delay time until request is recognized + maximum transaction response time < service deadline
- Time
    - Generally a periodic time-triggered schedule
    - Time line divided into a sequence of granules (cycle time)
    - Only one interrupt, a periodic clock interrupt for the start of a new granule
    - In distributed systems, synchronized to a precision of less than a granule
- Periodic with p_i being a multiple of the basic granule
- Schedule period = least common multiple of all p_i
- All scheduling decisions made at compile-time and executed at run-time
- Optimal schedule in a distributed system => NP complete

  Comparisons

- Predictability
    - Static Scheduling
        - Accurate planning of schedule, so precise predictability
    - Dynamic Scheduling

        - No schedulability tests exist for distributed system with mutual exclusion and precedence relations
        - Dynamic nature can not guarantee timeliness
- Testability
    - Static Scheduling
        - Performance tests of every task can be compared with established plans
        - Systematic and constructive since all input cases can be observed
    - Dynamic Scheduling
        - Confidence of timeliness based on simulations
        - Real loads not enough since rare events don't occur often
        - Are the simulated loads representative of real loads?
- Resource Utilization
    - Static Scheduling
        - Planned for peak load with time for each task at least the maximum execution time
        - If many operating modes, can lead to "combinatorial explosion" of static schedules
    - Dynamic Scheduling
        - Processor available more quickly
        - Resources needed to do dynamic scheduling
- Resource Utilization
    - Dynamic Scheduling (Cont'd)
        - If loads low, better utilization than static schedule
        - If loads high, more resources used for dynamic scheduling and less for execution of tasks
- Extensibility
    - Static Scheduling

- If a new task is added or the maximum execution time is modified, the schedule needs to be recalculated
- If the new node sends information into the system, the communications schedule needs to be recalculated
- Impossible to calculate static schedule if the number of tasks changes dynamically during run-time

- Dynamic Scheduling
  - Easy to add/modify tasks
  - Change can ripple through system
  - Probability of change and system test-time are proportional to tasks.
  - Assessing the consequences increase more than linearly with the number of tasks
  - Scales poorly for large applications

Speculative task

- **Speculative execution** is an optimization technique where a computer system performs some task that may not be actually needed. The main idea is to do work *before* it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work *after* it is known whether it is needed. If it turns out the work was not needed after all, any changes made by the work are reverted and the results are ignored.
- The objective is to provide more concurrency if extra resources are available. This approach is employed in a variety of areas, including branch prediction in pipelined processors, prefetching memory and files, and optimistic concurrency control in database systems.
- Modern pipelined microprocessors use speculative execution to reduce the cost of conditional branch instructions using schemes that predict the execution path of a program based on the history of branch executions. In order to improve performance and utilization of computer resources, instructions can be scheduled at a time when it has not yet been determined that the instructions will need to be executed, ahead of a branch.
- In compiler optimization for multiprocessing systems, speculative execution involves an idle processor executing code in the next processor block, in case there is no dependency on code that could be running on other processors. The benefit of this scheme is reducing response time for individual processors and the overall system. However, there is a net penalty for the average case, since in the case of a bad bet, the pipelines should be flushed. The compiler is limited in issuing speculative execution instruction, since it

requires hardware assistance to buffer the effects of speculatively-executed instructions. Without hardware support, the compiler could only issue speculative instructions which have no side effects in case of wrong speculation.

Collaborative Real Time Editing

What is it?

Collaborative editing is the practice of a group of individuals simultaneously editing a document. Using collaborative editing tools, authorized users can edit a document, see who else is working on it, and watch—in real time—as others make changes. Unlike simple version control, in which a single working copy of a file is managed among editors one at a time, collaborative editing allows multiple users to make changes at the same time. A group of individuals—in the same location or geographically separated—can use collaborative editing tools to create a document that reflects the contributions of the group, without having to track and coordinate edits. Collaborative documents are similar to wikis in that multiple users can change, add to, and delete content. They also resemble instant messaging in that users can see the input of all other users immediately. Some collaborative editing tools include instant messaging features so users can communicate in a chat session parallel to the document they are editing.

Who's doing it?

Collaborative editing was conceived as a tool for software developers, providing a way for two or more programmers to write code together, cross-checking each other's work and brainstorming how the application should work. Today, collaborative editing tools are being used more broadly. Authors co-writing a text can use collaborative editing to streamline the process of creating and revising content. A group of attendees at a workshop can write a single set of notes that are more complete than an individual could write alone. Similarly, some meeting organizers have come to rely on collaborative editing tools. Prior to a meeting, the leader writes an agenda. During the meeting, attendees use collaborative editing tools to access the agenda, update it with information they will present, and take notes on topics as they are covered. Some educators are using collaborative editing as a demonstration tool. For example, an instructor can put a document online where the students in a class can access it. The instructor assigns read-only rights to the students, who watch as the instructor edits or updates the document, demonstrating a method of proper revision. The instructor then allows each member of the class to edit the document in turn, while the other members
of the group watch.

How does it work?

With collaborative editing, a user creates a document and announces to other eligible users—who are often on the same subnet—that it is available for editing. Depending on how each document is set up, some users can see but not edit it, while others have full access. Many applications are built with "zero-configuration"

technology, which automatically locates users on a local area network and connects them without any input from the users; with other tools, users must manually add themselves. A window identifies users who are connected to the open document and assigns each a unique color. Each user's cursor and all of his or her edits are highlighted with that color, and all edits are displayed immediately, allowing everyone participating to see who does what, as it happens. The real-time nature of collaborative editing prevents simultaneous edits from overwriting one another, which, though uncommon, is possible with wikis. It is possible to change another author's text, but because editors can see changes instantly, the revision process is truly collaborative.

Why is it significant?

Collaborative editing is a more efficient method of creating and revising documents. Before collaborative editing, a group of users who all needed to participate in a document had to coordinate their editorial steps, maintaining control of several versions of the document to ensure the integrity of the changes. In addition, such processes typically require a project leader to coordinate different rounds of editing and resolve conflicts. With collaborative editing, all of these steps happen simultaneously. Contributors can see edits as they are made, saving time and eliminating the possibility that edits could be inadvertently overwritten or that conflicting edits will be resolved improperly. As a functional hybrid of wikis and instant messaging, collaborative editing creates a new dynamic for group work, whether for a formal paper or a set of lecture notes. Working simultaneously on a document can build a sense of community among the editors that is not possible if the document were simply passed from one individual to the next.

What are the downsides?

Effective collaborative editing depends on a conscientious document owner and a trusted group of editors. Although collaboratively edited files are stored on the document owner's computer, participating editors can make local copies of the file, which can result in the document's "splintering" into several versions. If the original of a document is closed during an editing session, the "master" can be handed off to any of the other users. Further editing of the document could result in lost edits and versioning problems. Editing tools also do not store a history of changes. Although each editor's changes are color-coded during editing, no record of the changes is stored—just the text is saved. The only way to preserve a snapshot of a document showing current edits is to take a screen shot or, with some applications, to export the document to a PDF.

Most collaborative editing applications are platform-specific, limiting their usefulness to people on different kinds of systems. The tools most commonly used by those with Macintosh operating systems are not compatible with either Linux- or Windows-based computers. Similarly, Windows-based tools generally do not work with Apple computers. These platform compatibility issues represent perhaps the biggest obstacle to broader adoption of collaborative editing. Because collaborative editing puts substantial control into the hands of the editors, any one of whom could sabotage an

entire document very easily, the practice requires high levels of respect and trust among the editors. In addition, some users may be uncomfortable moving editing and revision into a quasi-public,
shared space.

Where is it going?

Makers of collaborative editing applications are beginning to address cross-platform compatibility. Some tools in development are Web-based, requiring only a browser and Internet access. As tools become increasingly platform-independent, the opportunities for collaborative editing among wider-ranging groups of people will expand significantly. Collaborative editing tools are simple to use and are particularly well suited for students working on a variety of projects. Other developments are likely to include versioning, which allows users to see a history of changes and who made them, and further refining of protocols, both explicit and implied, that establish expectations for appropriate and acceptable editing.

What are the implications for teaching and learning?

Group work and multitasking are fast becoming two of the hall marks of today's learners, and collaborative editing is a natural fit for environments that support learning activities with technology. As students are increasingly asked to complete group assignments, collaborative editing tools offer an efficient way to accomplish them while taking part in an activity that many see as fun and that encourages them to engage with the material. Students in a large lecture or a small class can benefit from using collaborative editing tools to take "community notes"—reinforcing the material covered, seeing what other students identify as valuable, and developing a shared sense of respect among participants. Collaborative editing has special applicability for distance learning, where students may be separated by time and space but are still expected to work together on projects and to develop a sense of community.

Version Control Systems

- A component of software configuration management, **version control**, also known as **revision control** or **source control,** is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.
- The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and

complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

- **Version control systems** (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets and in various content management systems. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.
- Software tools for revision control are essential for the organization of multi-developer projects.

Structure

Revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by *checking out,* this is not in general immediately reflected in the revision control system (in the *repository*), but must instead be *checked in* or *committed.* A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving. Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer; in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the *repository,* and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and

data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

Source-management models

Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

Atomic operations

An operation is *atomic* if the system is left in a consistent state even if the operation is interrupted. The *commit* operation is usually the most critical in this sense. Commits tell the revision control system to make a group of changes final, and available to all users. Not all revision control systems have atomic commits; notably, CVS lacks this feature.

File locking

The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).

File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

Version merging

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.

Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text based documents, unless a specific merge plugin is available for the file types.

The concept of a *reserved edit* can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

Baselines, labels and tags

Most revision control tools will use only one of these similar terms (baseline, label, tag) to refer to the action of identifying a snapshot or the record of the snapshot. Typically only one of the terms *baseline*, *label*, or *tag* is used in documentation or discussion[]; they can be considered synonyms.

In most projects some snapshots are more significant than others, such as those used to indicate published releases, branches, or milestones.

When both the term *baseline* and either of *label* or *tag* are used together in the same context, *label* and *tag* usually refer to the mechanism within the tool of identifying or making the record of the snapshot, and *baseline* indicates the increased significance of any given label or tag.

Communication

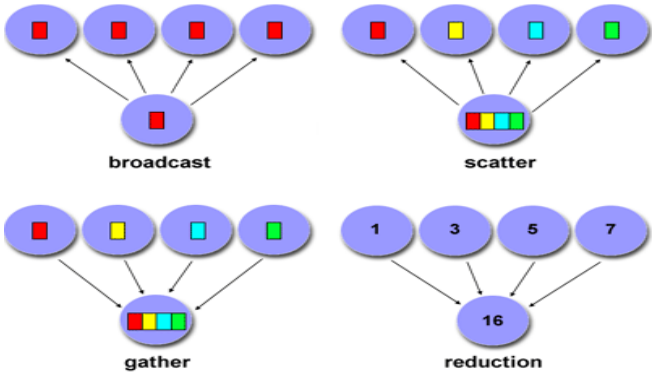
- Who needs Communications :
  - You don't need :
    - Some types of problems can be decomposed and execute in parallel. *Embarrassingly parallel*.
    - Very little inter-task communication is required
    - Eg. Image processing operation, every pixel in a black and white image needs to have its color reversed
  - You do need
    - Most parallel applications do require to share data with each other. (Eg. Ecosystem)
- There are a number of important factors to consider when designing program's inter-task communications:
  - Cost of communications
  - Latency vs. Bandwidth
  - Visibility of communications
  - Synchronous vs. Asynchronous communication
  - Scope of communications
  - Efficiency of communications
- Inter-task communication virtually always implies overhead
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.

- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems
- ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.
- Message passing Model: communications are explicit (under control of the programmer)
- Data Parallel Model: communications occur transparently to the programmer, usually on distributed memory architectures.

Communications (Synchronous vs. Asynchronous)

- Synchronous requires some type of "handshaking" between task that are sharing data.
- Synchronous : Blocking communications
- Asynchronous allow tasks to transfer data independently from one another.
- Asynchronous: Non-Blocking communications
- Interleaving computation with communication is the greatest benefit.

Communications (Scope-Collective)



Efficiency of communications

- Very often, the programmer will have a choice with regard to factors that can affect communications performance.
- Which implementation for a given model should be used? (Eg.MPI implementation may be faster on a given hardware platform than another)
- What type of communication operations should be used? (Eg. asynchronous communication operations can improve overall program performance)
- Network media - some platforms may offer more than one network for communications.

Synchronization (Types)

- Barrier
  - All tasks are involved
  - Each task perform its work. When the last task reaches the barrier, all task are synchronized
- Lock / semaphore
  - Typically used to serialize access to global data or section of code. Task must wait to use the code
- Synchronous communication operations
  - Involves only those tasks executing a communication operations (handshaking)

Message Passing

Non blocking Communication
• Advantages:
--
allows the separation between the initialization of the
communication and the completion.
--
can avoid deadlock
--
can reduce latency by posting receive calls early
• Disadvantages:
--
complex to develop, maintain and debug code

Non block Send/ Recv Syntax

• int MPI_Isend (void* message /* in */, int count /* in */, MPI_Datatype datatype /* in */, int dest /* in */, int tag /* in */, MPI_Comm comm /* in */, MPI_Request * request /* out */)

• int MPI_Irecv(void * message /* out */, int count /* in */, MPI_Datatype datatype /* in */, int source /* in */, int tag /* in */, MPI_Comm comm /* in */,
MPI_Request * request /* out */)

Non blocking Send/ Recv Details
• Non blocking operation requires a minimum of two function calls: a call to start the operation and a call to complete the operation.
• The “request” is used to query the status of the communicator or to wait for its completion.
• The user must NOT overwrite the send buffer until the send (data transfer) is complete.
• The user can not use the receiving buffer before the receive is complete.

Non blocking Send/ Recv Communication Completion
• int MPI_Wait (MPI_Request * request /* in - out */, MPI_Status * status /* out */)
• int MPI_Test (MPI_Request * request /* out */, int * flag /* out*/, MPI_Status * status /* out */)

• Completion of a non blocking send operation means that the sender is now free to update the send buffer “message”.
• Completion of a non - blocking receive operation means that the receive buffer “message” contains the received data.

Details of Wait/Test
• “ request ” is used to identify a previously posted send/receive
• MPI_Wait() returns when the operation is complete, and the status is updated for a receive.
• MPI_Test() returns immediately, with “flag” = true if posted operation corresponding to the “request” handle is complete.

Non - blocking Send/ Recv Example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv )
{
int my_rank , nprocs ,recv_count;
MPI_Request request;
MPI_Status status;
double s_buf [100], r_buf [100];
MPI_Init(&argc , & argv );
MPI_Comm_rank(MPI_COMM_WORLD, & my_rank );
MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
if (my_rank ==0 )
{
MPI_Irecv (r_buf , 1oo, MPI_DOUBLE, 1, 22, MPI_COMM_WORLD, &request);
MPI_Send(s_buf, 100, MPI_DOUBLE, 1,10, MPI_COMM_WORLD);
MPI_Wait(&request, &status);
}
else if(my_rank== 1)
{
MPI_Irecv(r_buf,1oo, MPI_DOUBLE,0, 10,MPI_COMM_WORLD, &request);
MPI_Send(s_buf, 100, MPI_DOUBLE,0,22,MPI_COMM_WORLD);
MPI_Wait(&request, &status);
}
MPI_Get_count((&status,MPI_DOUBLE,&recv_count);
printf(“proc%d, source %d, tag %d, count %d\n”, my_rank,status.MPI_SOURCE,
status.MPI_TAG, recv_count);
MPI_Finalize();
}
```

MPI collective Communications
•Routines that allow groups of processes to communicate.

• Classification by Operation:
– One – To - All Mode
One process contributes to the results. All processes receive the result.
MPI_Bcast()
MPI_Scatter(),
MPI_Scatterv()

–All-To-One Mode
All processes contribute to the result. One process receive the result.
•MPI_Gather(),
MPI_Gatherv()
•MPI_Reduce()

–All-To - All Mode
All processes contribute to the result. All processes receive the result.
•MPI_Alltoall(),
MPI_Alltoallv()
•MPI_Allgather(),
MPI_Allgatherv()
•MPI_Allreduce(),
MPI_Reduce_scatter()

Other Collective operations that do not fit into above categories
• MPI_Scan()
•MPI_Barrier()

Synchronous and Asynchronous Scheduling

Data transfers can be **synchronous** or **asynchronous**. The determining factor is whether the entry point that schedules the transfer returns immediately or waits until the I/O has been completed.

The read and write entry points are synchronous entry points. The transfer must not return until the I/O is complete. Upon return from the routines, the process knows whether the transfer has succeeded.

The aread and awrite entry points are asynchronous entry points. Asynchronous entry points schedule the I/O and return immediately. Upon return, the process that issues the request knows that the I/O is scheduled and that the status of the I/O must be determined later. In the meantime, the process can perform other operations.

With an asynchronous I/O request to the kernel, the process is not required to wait while the I/O is in process. A process can perform multiple I/O requests and allow the kernel to handle the data transfer details. Asynchronous I/O requests enable applications such as transaction processing to

use concurrent programming methods to increase performance or response time. Any performance boost for applications that use asynchronous I/O, however, comes at the expense of greater programming complexity.

Resource Management

Resource management is the efficient and effective deployment and allocation of an system's **resources** when and where they are needed. Such **resources** may include CPU, memory, etc.

Resource Requirements

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Availability
The ratio of the total time a functional unit is capable of being used during a given interval to (b) the length of the interval.

The most simple representation for **availability** is as a ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and down time, or

$$A = \frac{E[\text{uptime}]}{E[\text{uptime}] + E[\text{downtime}]}$$

Reliability

- Generally defined as the ability of a product to perform as expected over time
- Formally defined as the probability that a product, piece of equipment, or system performs its intended function for a stated period of time under specified operating conditions
- Reliability failure – failure after some period of use
- Reliability Measurement
  Failure rate (l) – number of failures per unit time

Scalability

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Fault Tolerance

This provides redundancy in case one component should fail, and also allows automatic error detection and error correction if the results differ. These methods can be used to help prevent single event upsets caused by transient errors. Although additional measures may be required in embedded or specialized systems, this method can provide a cost effective approach to achieve n-modular redundancy in commercial off-the-shelf systems.

Recovery

Data recovery is a process of salvaging inaccessible data from corrupted or damaged secondary storage, removable media or files, when the data they store cannot be accessed in a normal way. The data is most often salvaged from storage media such as internal or external hard disk drives (HDDs), solid-state drives (SSDs), USB flash drives, magnetic tapes, CDs, DVDs, RAID subsystems, and other electronic devices. Recovery may be required due to physical damage to the storage device or logical damage to the file system that prevents it from being mounted by the host operating system (OS).

The most common data recovery scenario involves an operating system failure, malfunction of a storage device, accidental damage or deletion, etc. (typically, on a single-drive, single-partition, single-OS system), in which case the goal is simply to copy all wanted files to another drive. This can be easily accomplished using a Live CD, many of which provide a means to mount the system drive and backup drives or removable media, and to move the files from the system drive to the backup media with a file manager or optical disc authoring software. Such cases can often be mitigated by disk partitioning and consistently storing valuable data files (or copies of them) on a different partition from the replaceable OS system files.

Protection – Using Locks and semaphores

All the methods for synchronization and serialization use the same underlying idea. They use variables in shared state as *signals* that all the processes understand and respect. This is the same philosophy that allows computers in a distributed system to work together --

they coordinate with each other by passing messages according to a protocol that every participant understands and respects. Locks, also known as *mutexes* (short for mutual exclusions), are shared objects that are commonly used to signal that shared state is being read or modified. For a lock to protect a particular set of variables, all the processes need to be programmed to follow a rule: no process will access any of the shared variables unless it owns that particular lock. In effect, all the processes need to "wrap" their manipulation of the shared variables in acquire() and release() statements for that lock.