Unit 6 : HPC Applications

Issues in sorting on parallel computers

Types of sorting on parallel processors, there are many issues to be addressed.

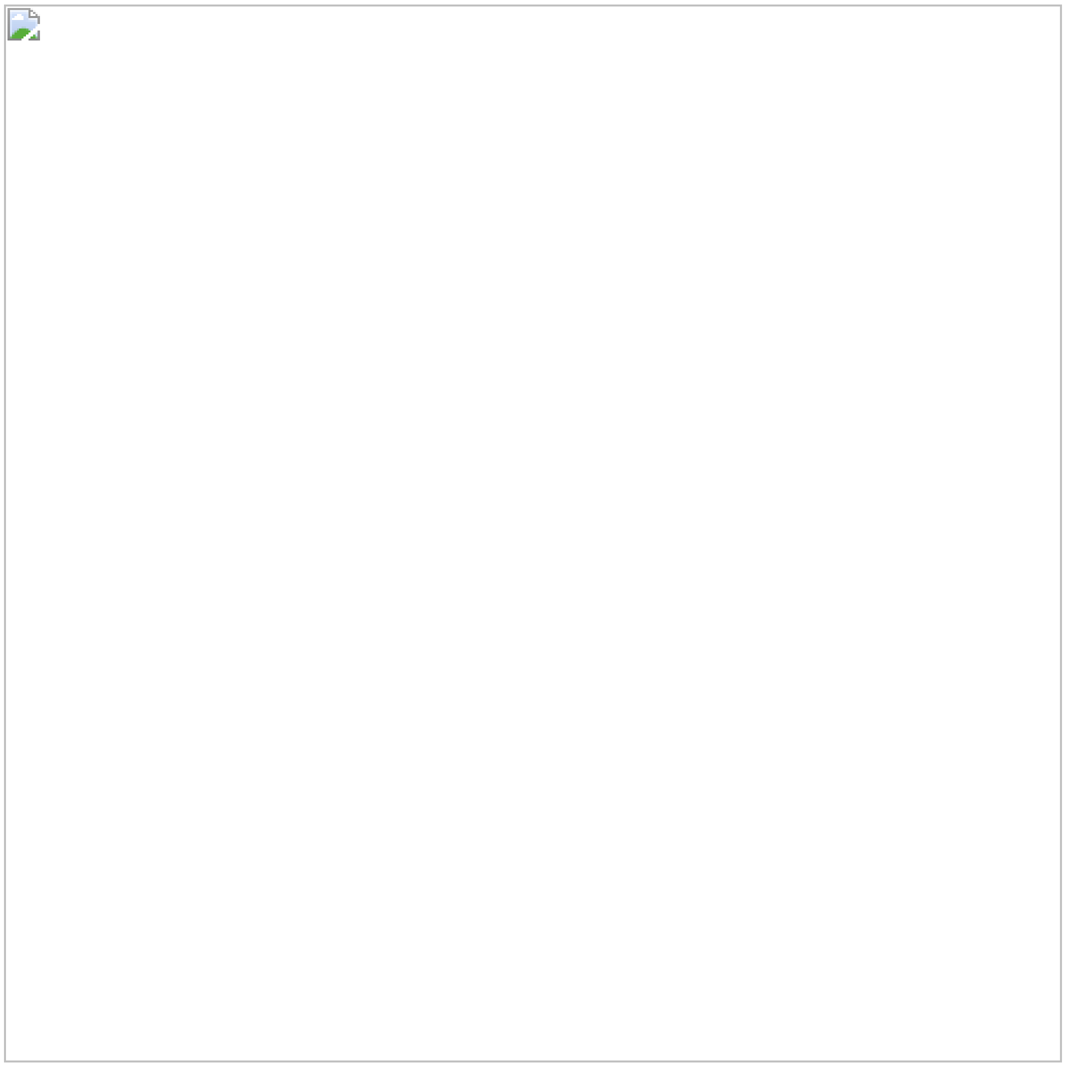
2 issues → where are the inputs and outputs stored, and how are comparisons performed

Input and output →

1. Sorted sequences are stored in the memory in sequential algorithms.
2. 2 places where input and output are stored : one of the processes, distribute among the processes and store on each process.

Comparisons →

1. Comparison based or non comparison based
2. sequential sorting can easily perform compare-exchange on two elements.
3. 1 element per process in comparison? one process keeps min and other process keeps max



4. If more than 1 process exists, each processor is assigned a block of n/p elements. (n elements and p processes)
5. compare and split operation →

Depth First Search algorithms

1. DFS expands the most recently generated nodes, and if it doesnt lead to any solution, it backtracks and expands a different node.
2. Major advantage → Storage requirement is linear in depth of the state space being searched.

3 algorithms:

1. Simple backtracking → DFS method that terminates on finding the first solution (greedy method) , no heuristic information is used. In ordered backtracking, the heuristics are used to order the successors of an expanded node.
2. DFBB (Depth first branch bound) → Continues to search even after finding a solution. If a new solution path is found, the best solution is updated, and whose extensions are guaranteed to be worse than the current best.
3. Iterative Deepening A* → Instead of searching deeper into a search space, we impose a bound on the depth of DFS algorithm.

If the node is beyond the depth, the node is not expanded and the algorithm backtracks.

If the solution is not found, entire search space is searched again using large depth bound.

Not guaranteeing the least cost path.

DFS 8 puzzle (refer book for now)

Best First Search

1. Uses heursitics to directly search to portions of search space to yield solutions.
2. Algorithm:
3. Smaller heuristics → promising nodes
4. 2 lists → open and closed
5. Initial node → open list, list is sorted according to a heuristic evaluation function that measures how likely each node is to yield a solution
6. The most promising node of the open list is removed and its expanded.
7. If this node is a goal node, the algorithm terminates otherwise the node is expanded.

8. Expanded node is in the closed list, and the successors of the new expanded node are in the open list.

Common BFS technique is A* algorithm

- The main drawback of any BFS algorithm is that its memory requirement is linear in the size of the search space explored.

Parallel Depth First Search

- Root node is expanded into nodes A and B, each assigned to a different processor.
- Each processor will independently search subtrees rooted at its assigned node.
- Imbalance in workload occurs if one subtree has significantly fewer nodes to expand, leading to idle time for that processor.
- The imbalance worsens with an increase in the number of processors.
- For example, with four processors, nodes A and B are expanded into nodes C, D, E, and F. If node E does most of the work, processors working on nodes C and D will be idle most of the time.
- Static partitioning of unstructured trees can lead to poor performance due to variance in the size of partitions rooted at different nodes.
- As the search space is usually generated dynamically, it is hard to estimate the size beforehand.
- Therefore, it is necessary to dynamically balance the search space among processors.

1. **Task Decomposition**: The search space is divided into smaller subspaces, and each subspace is assigned to a different processing unit (e.g., CPU core or GPU thread). This division can be done statically or dynamically, depending on the architecture and the problem characteristics.
2. **Traversal**: Each processing unit independently traverses its assigned subspace using the DFS strategy. It starts from a given initial node and explores adjacent nodes recursively, following a depth-first traversal order.
3. **Communication and Synchronization**: Depending on the specific parallelization strategy, processing units may need to communicate or synchronize their progress. For example, when a processing unit encounters a boundary between its assigned subspace and that of another unit, it may need to exchange information about the visited nodes or synchronize to avoid redundant exploration.

4. **Load Balancing**: Ensuring load balance among processing units is crucial to achieving efficient parallelization. Load balancing strategies may involve dynamically redistributing workload among processing units to prevent some units from finishing their work significantly earlier than others.
5. **Termination Condition**: The parallel DFS terminates when all processing units have completed their assigned subspace exploration. This may involve aggregating the results obtained by each unit or determining a global termination condition based on the problem requirements.

Dynamic load balancing

1. If processor runs out of work it gets more work from another processor that has work.
2. When a processor finishes searching its part of search space, it requests unsearched part from other processors.
3. Each processor has a local stack → for executing DFS.
4. When its local stack is empty, it requests untried alternatives from another processors stack.

Work Splitting strategies → Donors stack is split into 2 stacks, one of which is sent to the recipient. Ideally the stack is split into 2 equal pieces, such that the search space is the same.

Load Balancing Schemes → Asynchronous round robin, each processor maintains an independent variable. Whenever a processor runs out of work, it uses target as the label of a donor processor and attempts to get work from it.

Random polling → Simplest load balancing scheme, when a processor becomes idle, it randomly selects a donor.

Parallel Best-first search

1. In most parallel formulations of BFS, different processors concurrently expand different nodes from the open list.
2. Given p processors, the simplest strategy assigns each processor to work on the current best nodes on the open list. This is called as centralised strategy.

Problems of this approach:

1. **Potential Overexpansion**: One issue with this parallel formulation is the potential overexpansion of nodes. If the first node on the open list is a solution, the parallel formulation may expand the first p nodes, where p is the number of processors. However, because it always picks the best p nodes, the extra work is limited.
2. **Termination Criterion**: In sequential BFS, the termination criterion relies on finding the best goal node, ensuring that the search terminates when the best solution is found. However, in parallel BFS, due to concurrent node expansion, the termination criterion may fail because a solution found by one processor may not correspond to the best goal node overall. The termination criterion needs modification to ensure termination only after the best solution has been found.
3. **Open List Accessibility**: Accessibility to the open list poses another challenge in parallel BFS. Since the open list is accessed for each node expansion, it must be easily accessible to all processors. Even on shared-address-space architectures, contention for the open list can limit performance, impacting speedup.

1. One way to avoid the prb due to a centralised open list → each processor should have a local open list.
2. All processors then select and expand nodes simultaneously.
3. Processors must communicate with each other to minimize unnecessary search.

Communication strategies

Random Communication strategy

1. In random communication strategy → Each processor periodically sends some of its best nodes to the open list of a processor.
2. Each processor periodically sends some of its best nodes to the open list of a random processor.
3. The main objective of this strategy is to ensure that if one processor discovers a promising part of the search space, the other processors also benefit from it. By sharing information in this manner, the entire parallel BFS algorithm can collectively explore promising regions of the search space more efficiently.

Ring Communication Strategy

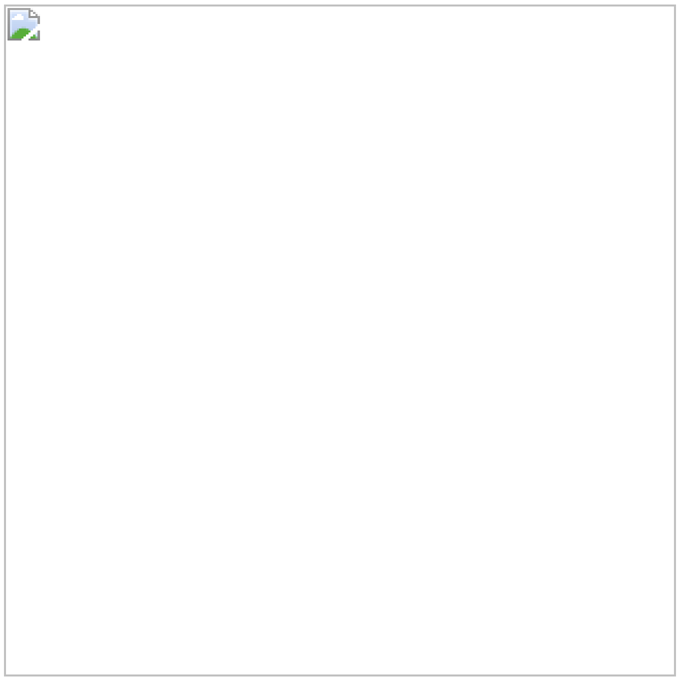
1. Processors are mapped in a virtual ring.
2. Each processor periodically exchanges some of its best nodes in the open list.

Blackboard communication strategy

1. Shared blackboard is present through which nodes are switched among processors.
2. The node is exchanged only if its heuristic value is within tolerable limit of the best node on the blackboard.

Bubble sort and its variants

1. Bubble sort → swapping adjacent elements if not in the right order till no swaps are needed.



2. Time complexity → O (n^2)
3. 2 variants of bubble sort that give parallelisation
    a. odd even transposition
    b. shell sort
4. Odd even transposition
    a. Sorts n elements in n phases.
    b. n/2 compare exchange operations.
    c. This algorithms alternates between phases, odd and even.
    d. Odd phase, elements of odd indices are compared with right neighbours, (a1,a2) (a3,a4)…etc.
    e. Similarly, during even phase, elements with even indices are compared with their right ones, (a2,a3) etc.
    f. Drawback : Few elements out of order, need o(n^2)
    g. Parallel runtime - o(n)
5. Shell sort
    a. Shell sort compares distant elements rather than adjacent.

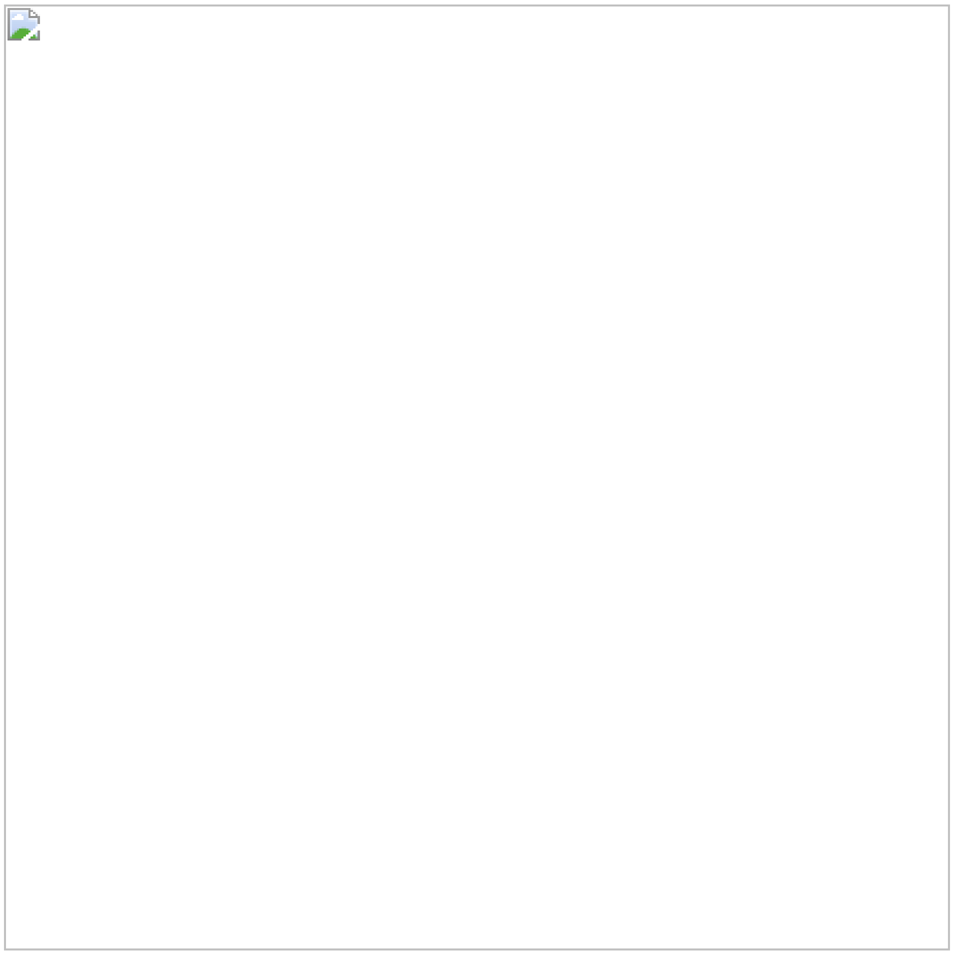
b. Gap is the spacing between elements.

c. Gap is reduced by 1 till we reach the last pass.

d. interval = floor (n/2)

Parallel Shell sort:

1. First phase → processes that are far, compare-split their elements.
2. Second phase → odd even transposition
3. Parallel shell sort makes parallel odd even transposition faster.



4. In short :

    a. N elements, P processors,
    b. Each processor gets N/P elements.
    c. N/P elements in each processor undergo shell sort (describe about shell sort)
    d. Compare and split takes place → farther groups come together
    e. Odd even transposition takes place.

Parallel shell sort speeds up odd even transposition.

Parallelising Quick sort

Sequential quick sort →

1. Selection of pivot.
2. Dividing the list → numbers lesser than the pivot and higher than the pivot.
3. Final result → concatenation of sorted low lists, pivot and sorted high lists.

Parallel Quicksort →

1. Each process holds a segment of unsorted list.
2. Randomly → pivot is chosen, and broadcasted to every process.
3. Each process → divides its list into 2 lists → smaller than pivot and greater than pivot.
4. Each process in upper half of the list sends its low list to a partner process in the lower half of the process list and recieves a high list in return.
5. Upper half → values greater than the pivot
6. Lower half → values lesser than the pivot
7. Processes are divided and recursive approach is followed.

Distributed Computing

Document Classification

1. Document classification → Process of assigning predefined labels or categories to the documents which include unstructured or semi structured information
2. Its the process of retrieving, filtering, clustering and extracting documents.
3. General procedure:
    a. Set of preclassified documents → training set
    b. Set is analysed in order to derive a classification scheme
    c. This is refined with a testing process
4. Parallel processing for document classification can take the advantage of increasing availability of multiprocessors or multi core processors.
5. GPU consists of a multiprocessor and they can perform parallel processes in small amount of time
6. Automatic document classification task:
    a. Document D belongs to category C, according to the knowledge of the correct categories known for the training documents.
    b. A single document may belong to more than 1 category and each category may contain more than 1 document.
    c. GPU takes advantage of large execution threads to find work when some of them are waiting for long.
    d. Small cache memories are provided to help control the bandwidth requirements.
    e. Executing the sequential parts → CPU
    f. Numerically intensive parts → GPU

g. Implementation of document classification using GPU reduces time complexity and testing huge set of documents.

h. Parallel KNN can be implemented in CUDA.

Kubernetes

1. Known as K8s is used for automating deployment, and scaling and management of containerised applications that support distributed computing.
2. Kubernetes offers portability, scalability, extensibility, adding end to end development, operations, security control.
3. Kubeflow → parallel computing for AI / ML
4. Kubeflow is a learning toolkit for K8s that maintains workflow on Kubernetes.
5. Provides straightfwd ways to deploy open source systems for ML to diverse infrastructures.
6. Supports Jupyter notebooks, Tensorflow model training to train model.
7. Kubeflow mission → Make scaling ML models and deploying them to production as simple as possible.
8. → Scaling based on demand
9. → Deploying and managing loosely coupled microservices.
10. → easy, repeatable and portable deployments.
11. Stages : Data preparation, Model training, Prediction serving, Service management
12. Comprehensive solution for handling end to end ML workflows

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

Kubernetes simplifies the management of complex distributed systems by providing a unified platform for container orchestration and management.

Key Features of Kubernetes:

1. **Container Orchestration**: Kubernetes automates the deployment, scaling, and management of containerized applications across clusters of machines. It abstracts away the underlying infrastructure and provides a consistent interface for deploying and managing applications.
2. **Automatic Scaling**: Kubernetes can automatically scale applications based on resource utilization or other metrics. It supports both horizontal scaling (adding or removing instances of an application) and vertical scaling (adjusting the resources allocated to each instance).
3. **Self-Healing**: Kubernetes monitors the health of applications and automatically restarts containers that fail. It can also replace containers that do not respond to health checks or meet defined criteria, ensuring high availability and reliability.
4. **Service Discovery and Load Balancing**: Kubernetes provides built-in mechanisms for service discovery and load balancing. It allows applications to discover and communicate with each other using DNS or environment variables, and it distributes incoming traffic across multiple instances of an application for improved performance and reliability.
5. **Rolling Updates and Rollbacks**: Kubernetes supports rolling updates and rollbacks of application deployments, allowing new versions of applications to be deployed gradually while minimizing downtime and impact on users. It also provides mechanisms for rolling back to previous versions in case of issues or failures.
6. **Storage Orchestration**: Kubernetes provides storage orchestration capabilities, allowing applications to dynamically provision and access storage resources such as persistent volumes and volumes backed by cloud providers or storage systems.
7. **Configuration Management**: Kubernetes supports declarative configuration management using YAML or JSON files. It allows users to define the desired

state of their applications and infrastructure, and Kubernetes takes care of implementing and maintaining that state.

Applications of Kubernetes:

1. **Microservices Architecture**: Kubernetes is well-suited for deploying and managing microservices-based applications, where each component of the application is packaged as a container and deployed independently.
2. **Continuous Integration and Continuous Deployment (CI/CD)**: Kubernetes can be integrated with CI/CD pipelines to automate the deployment of applications from development to production environments, enabling faster release cycles and improved developer productivity.
3. **Hybrid and Multi-Cloud Deployments**: Kubernetes provides a unified platform for deploying and managing applications across hybrid and multi-cloud environments, allowing organizations to avoid vendor lock-in and leverage the benefits of multiple cloud providers.
4. **Big Data and Machine Learning Workloads**: Kubernetes is increasingly being used to deploy and manage big data and machine learning workloads, providing scalable and reliable infrastructure for processing and analyzing large datasets.
5. **Edge Computing**: Kubernetes can be used to deploy and manage applications at the edge, closer to the point of data generation or consumption, enabling low-latency processing and improved user experiences in edge computing scenarios.m