Designing Parallel Algorithms

Parallel Algorithms

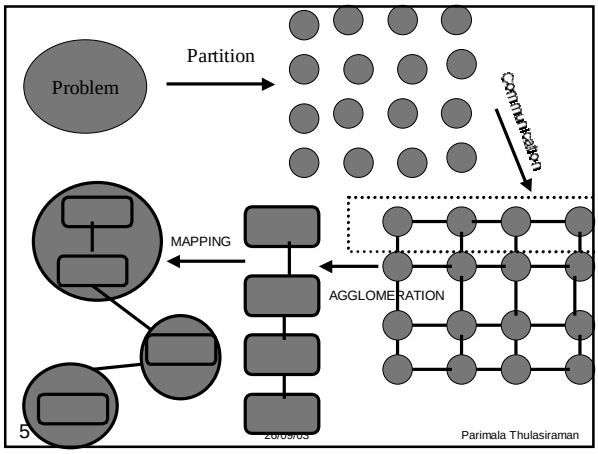
- 4 desirable attributes of Parallel Algorithm and Software
  - Concurrency : ability to perform many actions simultaneously; essential to execute programs on many processors
  - Scalability: increasing processor count
  - Locality : a high ratio of local memory access to remote memory access (communication) ; key to high performance on Multicomputer architectures
  - Modularity: decomposition of complex entities into simpler components (software engineering aspects)

Methodical Design

- Methodology
  - Concurrency (Machine-independent issues) first
  - Machine specific aspects of design later
- Design Process: (PCAM)
  - Partitioning
  - Communication } Concurrency and scalability
  - Agglomeration
  - Mapping } Locality and performance issues

Parallel Algorithms

- Partition Computations
  - Domain
  - Functional
- Communication structures
  - Local and global
  - Static and dynamic
  - Structured and unstructured
  - Synchronous and asynchronous
- Agglomeration: means of reducing communication and implementation costs
- Mapping



Four Stages

- Partitioning
  - Computation and data are decomposed into smaller tasks
  - Parallel execution (# of processors ignored)
- Communication
  - What type of communication required to coordinate task execution
  - Algorithms and communications structures are defined
- Agglomeration
  - Task and communication structures are evaluated w.r.t performance requirement and implementation costs
  - Tasks maybe combined into larger tasks to improve performance or reduce cost
- Mapping
  - Each task is assigned to a processor to maximize processor utilization, minimize communication costs
  - Mapping can be static or dynamic

Partitioning

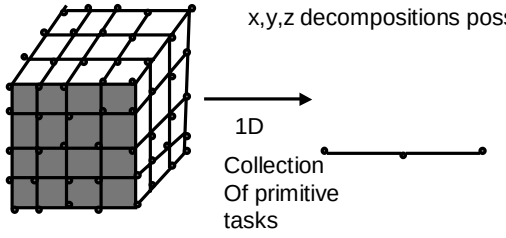
- Expose maximum parallelism
- Partition *computation* and *data*
- Define a large number of tasks : *fine grained* decomposition of a problem; provides greatest flexibility
- Programmers usually first focus on data, partition data and then associate computation with data : **domain decomposition**
- Algorithmic approach- **functional decomposition**
  - first decomposing the computation and
  - then dealing with data

Domain Decomposition

- Decompose data associated with problem
- Divide data into small pieces approximately of equal size
- Next partition computation
  - Yields a number of tasks, some data and a set of computations onthat data
  - An operation may require data from several tasks : communication required(addressed in the next phase)
- Data maybe input, output or intermediate values
- Different data structures permit different partitions
- First focus on the data structure that will be most frequently used; different phases of the algorithm may require different data structures
- Treat each phase separately; then determine how the decompositions and parallel algorithms fit together in each phase

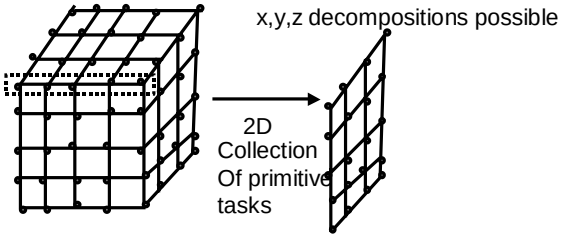
Domain Decomposition

- 3D grid



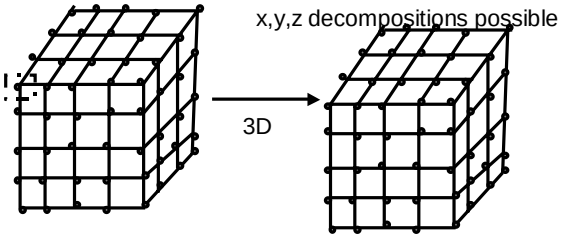
Domain Decomposition

- 3D grid



Domain Decomposition

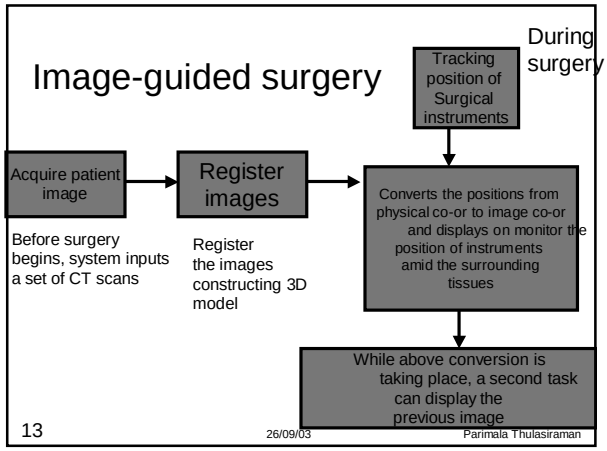
- Always best to maximize primitive tasks
- 3D partitioning is preferred
- 3D grid



Functional Decomposition

- Decompose computation into disjoint tasks
- Examine data requirements; data maybe disjoint
- Or data may overlap needing considerable communication (indication for Data Dependency)
- Yields a collection of tasks that achieve concurrency through pipelining

Image-guided surgery



Partitioning

- Domain and functional decompositions are 2 complementary techniques that may be applied to different components of a single problem or even applied to the same problem to alternative parallel algorithm
- Avoid replicating data and computation: partition both computation and data into disjoint sets
  - May have to at later stages to reduce communication

Partitioning

- Whichever strategy is used, we call each of these tasks a primitive task
- Goal: identify as many primitive tasks as possible
  - Upper bound on the parallelism we can exploit

Partitioning Design Checklist

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer?
  - If not, you have little flexibility in subsequent design stages
- Does your partition avoid redundant computation and storage requirements?
  - If not algorithm may not be scalable with large problems
- Are tasks of comparable size?
  - If not, hard to allocate equal amount of work to each processor
- Does the number of tasks scale with problem size? Increase in problem size should increase number of tasks rather than size of individual tasks.
  - If not, may not be able to solve larger problems when more processors are available
- Have you identified several alternative partitions?

Communication

- Direct link between consumer and producer
- Messages that travel over the link (sending and receiving)
- Avoid introducing unnecessary links and communication operations
- Seek to optimize performance by:
  - distributing communication operations over many task
  - Organizing communication operations that permit concurrent execution
- Domain decomposition: communication requirements difficult to predict
- Functional decomposition data flow approach between tasks

Communication Patterns

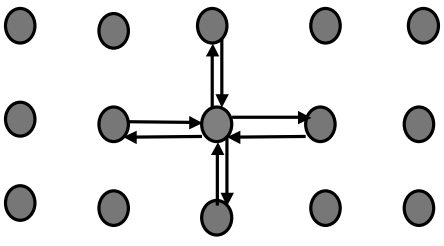
- **Local/Global**:
  - in local communication each task communicates with a small set of other tasks (its neighbors);
  - global communication requires each task to communicate with many tasks
    - Example : computing sum of the values stored in the processes
- **Structured/Unstructured**: in structured communication a task and its neighbors form a regular structure(tree,grid); unstructured communication networks maybe arbitrary graphs
- **Static/Dynamic**: in static communication neighboring points do not change over time; partners change over time in dynamic communication (determined at runtime)
- **Synchronous/Asynchronous**: producer and consumer are in synch. (coordinating fashion); in asynchronous communication, consumer obtains the data without the cooperation of the producer

Communication among tasks part of the overhead of Parallel algorithms→ sequential algorithms do not do this.

Local communication

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i-1,j}^{(t)} + X_{i,j+1}^{(t)}}{8}$$

Jacobi Finite Difference



For parallel computing : Jacobi preferred

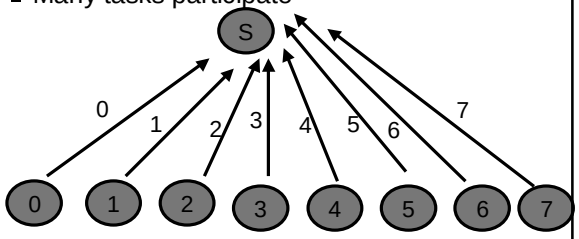
Choice of solution strategies important in parallel Computing performance

Global Communication

Parallel Reduction Operation

$$S = \sum_{i=0}^{N-1} X_i$$

- Many tasks participate



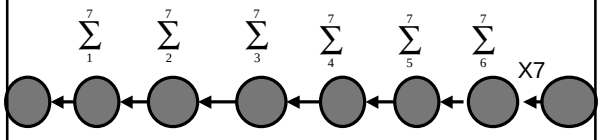
Assume manager receives values to sum

Global Communication

- Manager sums each number one at a time
- O(N) time; not a good parallel algorithm!!
- Problems with this approach
  - ***Centralized algorithm***: does not distribute computation and communication. A single task (S) must participate in every operation
  - ***Sequential***: does not allow multiple computation and communication operations to proceed concurrently

Distributing communication and computation

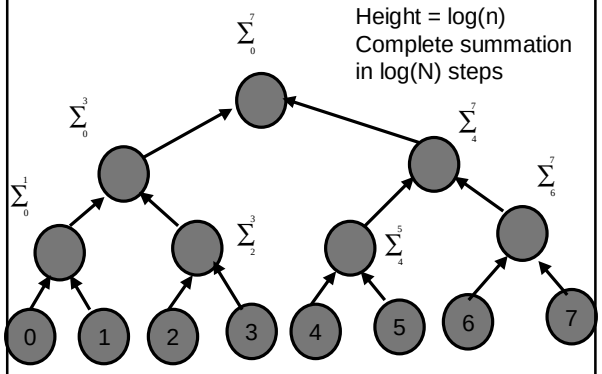
- Pipelining (good if multiple summation operations needed)



Divide and Conquer Strategy

- To solve a complex problem, partition it into 2 simpler problems of equal size
- Recursively apply the process to produce a set of subproblems that cannot be subdivided anymore

```
Procedure divide_and_conquer
Begin
        if base case then
                solve problem
        else
                partition problem into subproblems L and R
                Solve problem L using divide_and_conquer
                Solve problem R using divide_and_conquer
                Combine solutions to problems L and R
        endif
end
```



Unstructured and Dynamic Communication

- So far, considered regular, structured communication(tree)
- In some cases, communication is very complex
- Finite element methods: computational grid maybe shaped to follow an irregular object
  - Grid points irregular,
  - Data dependent
  - Partners change over time
- Agglomeration and mapping is a problem
- If communication is dynamic, these algorithms have to applied frequently, costly

Asynchronous Communication

- No global clock between producer and consumer
- Producers cannot predict when consumers require data
- Consumers explicitly require data from producers
- Shared memory machines, even more difficult; read and write must occur in proper order

Communication Design Checklist

- The communication operations are balanced among the tasks?
- Each task communicates with only a small number of neighbors?
- Can the tasks perform their communications concurrently?
- Can the tasks perform their computations concurrently?

Agglomeration

- Moving from abstract to concrete
- Decisions made in partitioning/communication can effectively execute on a parallel computer?
- Consider the parallel computer, the number of processors, should we *replicate* data or combine task done in partitioning
- Is it SPMD?

Agglomeration

- If the number of tasks exceeds the number of processors by several orders of magnitude, simply creating these tasks would be significant overhead.
- We consider how to combine these tasks.
- And map onto the processors to reduce the amount of parallel overhead.
- In MPI, we usually have one task per processor and mapping is very trivial.

Goals

- To lower communication overhead.
  - If we agglomerate the tasks that communicate with each other communication is completely eliminated.
    - Data values controlled by the primitive values are now in the memory of the consolidated task.
    - Called as increasing the locality of the parallel algorithm
- Maintain scalability of parallel design
  - Make sure we have not combined so many tasks that we are not able to port the program onto a future computer with more number of processors.
- Reduce s/w engineering costs
  - Parallelizing a sequential code, one agglomeration may make greater use of the sequential code, reducing time and expense of developing the parallel algorithm.

Agglomeration Design Checklist

- Has agglomeration reduced communication costs by increasing locality?
- If agglomeration has replicated computation, have you verified that the benefits of this replication outweighs its costs for a range of problem sizes and processor counts?
- If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
- Has agglomeration yielded tasks with similar computation and communication costs?
- Does the number of tasks still scale with problem size?
- If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers ?
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing s/w costs or reducing scalability?
- If you are parallel zing an existing sequential algorithm,have you considered the cost of the modifications required to the sequential code?

Mapping

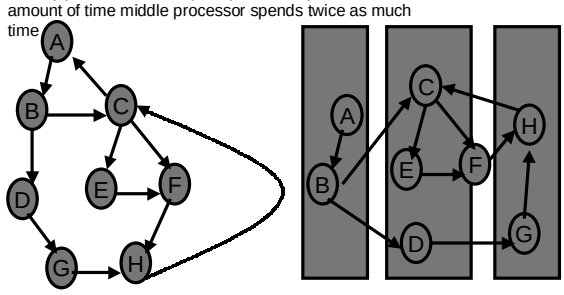
- Process of assigning tasks to processors
- On centralized multiprocessor,
  - OS automatically maps processes to processors
- So, we assume distributed memory machine

Goals

- Maximize processor utilization
- Minimize interprocessor communication
- Processor Utilization:
  - Average percentage of time the system's processors are actively executing tasks necessary for the solution of the problem
  - Maximized when the computation is balanced evenly allowing all processors to begin and end execution at the same time.
    - It drops when some processors are idle while others re busy.

Interprocessor communication

If every processor is of equal speed, every task same amount of time middle processor spends twice as much time



Mapping

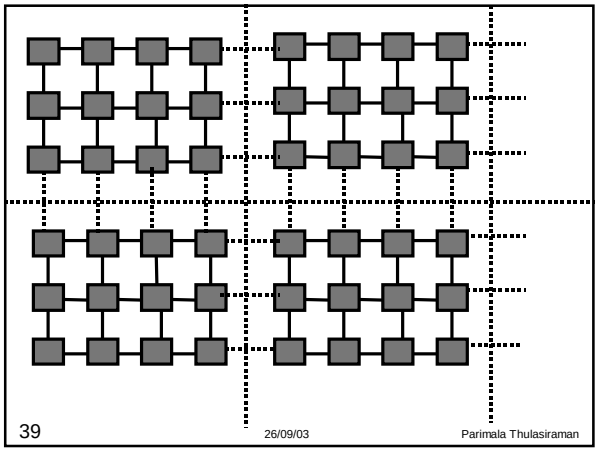
- We place tasks that able to execute concurrently on processors, so as to enhance concurrency
- We place tasks that communicate frequently on the processor, so as to increase locality
- These are conflicting strategies, have to deal with tradeoffs

For example

- If *p* processors, mapping every task to the same processor reduces interprocessor communication to 0.
- But, reduces utilization to *1/p*.
- Need to select a middle point
- Finding an optimal solution to mapping is NP-hard
  - NO POLYNOMIAL TIME ALGORITHM
  - ONLY HEURISTICS

Mapping

- Mapping problem is NP-complete: no polynomial time algorithm exists
- Heuristics and knowledge
- Domain Decomposition: number of fixed tasks/structured local.global communication
  - Mapping straightforward
  - Map to minimize interprocessor communication
  - Agglomerate tasks—one per processor
  - P coarse-grained tasks
- Complex domain decomposition: many work tasks/unstructured communication
  - Agglomeration and mapping strategies not efficient
  - Employ Load Balancing Algorithms



Load Balancing

- Load balancing algorithms identify agglomeration and mapping strategies through heuristics
  - Time required to execute these algorithm must be weighed against the benefits of reduced execution time
- **Probabilistic Load Balancing**: lower overhead than methods that exploit structure in an application
- **Dynamic load balancing**: employed periodically due to changes during runtime. Determine efficient agglomeration and mapping.
- **Functional Decomposition**: many short lived tasks that coordinate with other tasks only at the start and end of execution : task scheduling algorithms (tasks allocated to idle processors)

Load Balancing Algorithms(Based on Domain Decomposition)

- Recursive Bisection
- Local algorithms
- Probabilistic Methods
- Cyclic Mapping

} Partitioning algorithms

Fine grained tasks agglomerated into one coarse grained task per processor

Recursive Bisection

- Partition a domain(eg. Finite element grid) into subdomains of approximately equal computational cost while attempting to minimize communication costs (crossing task boundaries)
- Divide and Conquer approach
- Recursively cut these subdomains
- Notice that this partitioning algorithm itself can be executed in parallel

Recursive Bisection

- Recursive coordinate bisection: applied to irregular grids;
  - Cuts based on physical coordinates of grid points
  - Subdivides along longer dimensions
  - E.g.,cut made along x-dimension, grid points in one subdomain will have an x-coordinate greater than the grid points in the other.
  - Simple and inexpensive
  - does not optimize communication performance : can generate long skinny subdomains

Recursive Bisection

- Unbalanced recursive bisection: (Plate 1)
- Recursive graph bisection:
  - treats a grid as a graph with N vertices
  - Uses connectivity information to reduce the number of grid edges crossing subdomain boundaries to reduce communication
- Recursive spectral bisection(even better):Plate2

Local Algorithms

- Compensate for changes in computational load using only information obtained from a small number of neighboring processors
  - Eg. Periodically each processor (in a mesh) compares its computational load with that of its neighbors in the mesh and transfers computation if the difference in load exceeds certain threshold
  - Slow if major changes occur during the course of the execution (one processor is bombarded with many tasks)

Probabilistic Methods

- Allocate tasks to randomly selected processors
- If the number of tasks is large, each processor will be allocated same number of tasks (hopefully!!)
- Low cost and scalability
- Acceptable load distribution if there are more tasks than processors
- Little locality desired and little communication
- Disadv: More communication than other techniques

Cyclic Mapping

- P processors, each processor is allocated every Pth task.
- Allocated about the same computational load
- Improved load balance weighed against increased communication costs due to reduced locality
- Block cyclic: blocks of tasks are allocated to processors

Task Scheduling algorithms

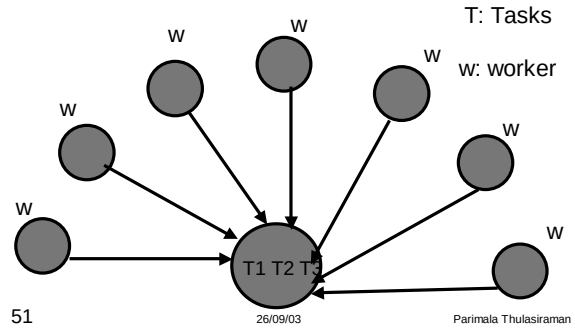
- Functional decomposition: many tasks
- Centralized or distributed task pool: new tasks are placed and from which tasks are taken for allocation to processors
- How to allocate tasks to workers(processors)?

Approaches

- Manager/Worker
- Hierarchical Manager/Worker
- Decentralized Schemes
- Termination detection

Manager/Worker

- A central manager task is given responsibility for problem allocation
- Each worker repeatedly requests and executes a problem from the manager
- Workers can also send tasks to the manager for allocation to other workers
- Efficiency:depends on number of workers and cost of executing and obtaining problems



Hierarchical Manager/Worker

- A set of workers have a submanger
- Workers request tasks from submanagers
- Submanagers communicate with managers and submanagers periodically to balance load between the sets of processors for which they are responsible

Decentralized Schemes

- No central manager
- A separate pool of tasks on each processor
- Idle processors request work from other processors
- Responsible for computing and managing the queue of tasks

Termination Detection

- Task scheduling algorithms require a mechanism for determining when a search is complete; otherwise idle workers will never stop requesting info. From other processors(workers)
- In centralized schemes: the manager determines when to end
- Decentralized: more difficult to detect termination (there may be tasks in transit when all processor may look idle)

Mapping Design Checklist

- If considering an SPMD design for a complex problem, have you considered an algorithm based on dynamic task creation and deletion?
- If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm
- If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck?
- If using a dynamic load balancing scheme, have you evaluated the relative costs of different strategies
- If using probabilistic or cyclic methods, do you have a large enough # of tasks to ensure reasonable load balance