CS6868 - Concurrent Programming

Design patterns for parallel programs

V. Krishna Nandivada

IIT Madras

A pattern language

- **Pattern**: “a careful description of a perennial solution to a recurring problem within a . . . context.”
- **Origin** Christopher Alexander, 1977 in the context of design and construction of building and town.
- Patterns in software engineering: Beck and Cunningham (1987), Gamma, Helm, Johnson, Vlissides (1995).
- **Pattern Language**: a structured method of describing good design practices within a field of expertise.

A pattern language for parallel programs

Finding Concurrency

Structure the given problem to expose exploitable concurrency.

Algorithm Structure

Structuring the algorithm to take advantage of potential concurrency.

Supporting Structures

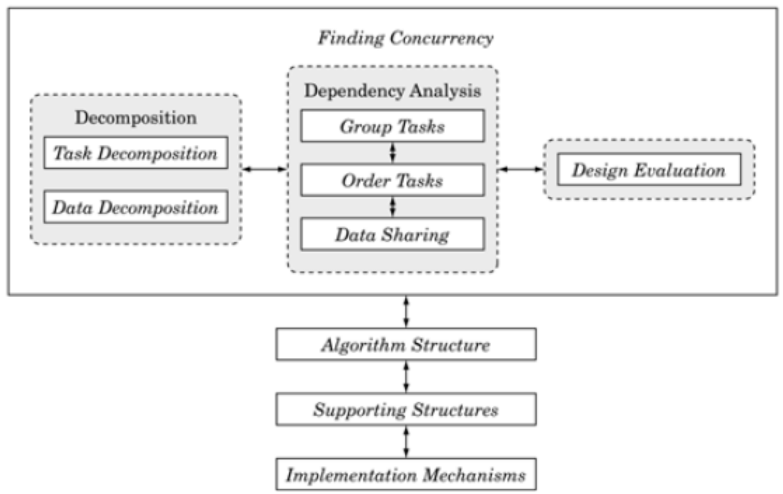
Implementation Mechanisms

Helps algorithm to be implemented.

How the high level specifications are mapped.

Goal: Identify patterns in each stage.

Finding concurrency in a given problem - deep dive



Decomposition Patterns

- Task decomposition: A program to a sequence of "tasks".
  - Some of the tasks can run in parallel.
  - Independent the tasks the better.
- Data decomposition: Focus on the data used by the program. Decompose the program into tasks based on distinct chunks of data.
  - Efficiency depends on the independence of the chunks.
- Task decomposition may lead to data decomposition and vice versa.

Q: Are they really independent?

Task decomposition: An approach

- Identify "resource" intensive parts of the problem.
- Identify different tasks that make up the problem. Challenge: write the algorithms and run the tasks concurrently.
- Sometimes the problem will naturally break into a collection of (nearly) independent tasks. Sometimes, not!
- Q: Are there enough tasks to keep the map all the H/W cores?
- Q: Does each task have enough work to keep the individual cores busy?
- Q: Are the number of tasks dependent or independent of the number of H/W core?
- Q: Are these tasks relatively independent?
- Instances of tasks: Independent modules, loop iterations.
- Relation between tasks and ease of programming, debugging and maintenance.

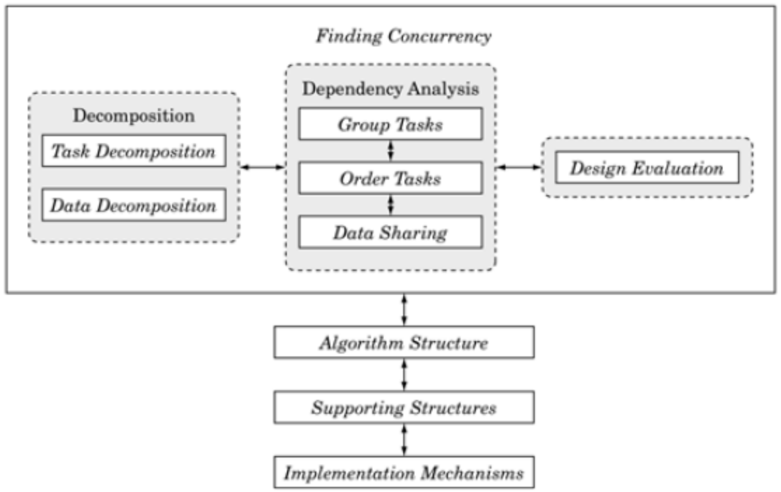
Task decomposition: Matrix multiplication example

$$C = A \times B$$

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}$$

- "Resource" intensive parts?
- Tasks in the problem?
- Are tasks independent? Enough tasks for all the cores? Enough work for each task? Size of tasks and number of cores?
- Each element $C_{i,j}$ is computed in a different task - row major.
- Each element $C_{i,j}$ is computed in a different task - column major.
- Each element $C_{i,j}$ is computed in a different task - diagonals.
- How to reason about Performance? Cache effect?

Finding concurrency in a given problem



Data decomposition: Design

- Besides identifying the "resource" intensive parts, identify the key data structures required to solve the problem, and how is the data used during the solution.
- Q: Is the decomposition suitable to a specific system or many systems?
- Q: Does it scale with the size of parallel computer?
- Are similar operations applied to different parts of data, independently?
- Are there different chunks of data that can be distributed?
- Relation between decomposition and ease of programming, debugging and maintenance.
- Examples:
  - Array based computations: concurrency defined in terms of updates of different segments of the array/matrix.
  - Recursive data structures: concurrency by decomposing the parallel updates of a large tree/graph/linked list.

Data decomposition: Matrix multiplication example

$$C = A \times B$$

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}$$

- "Resource" intensive parts?
- Data chunks in the problem?
- Does it scale with the size of parallel computers?
- Operations (Reads/Writes) applied on independent parts of data?
- Data chunks big enough to deem the thread activity beneficial?
- How to decompose?
- Each row/column of $C_{i,j}$ is computed in a different task.
- Each column of $C_{i,j}$ is computed in a different task.
- Performance? Cache effect?
- Note: Data decomposition also leads to task decomposition as well.

Matrix multiplication: Data decomposition.

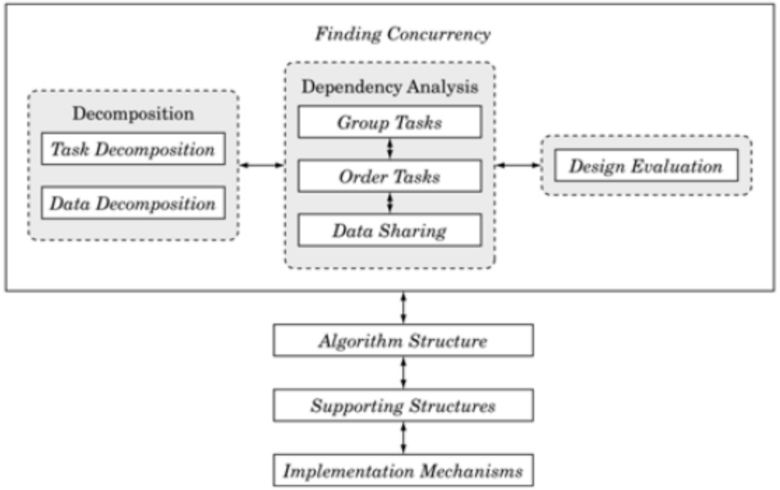
$$C = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

$$= \begin{pmatrix} A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{pmatrix}$$

Advantages

- Can fit in the blocks into cache.
- Can scale as per the hardware.
- Overlap of communication and computation.

Finding concurrency in a given problem



Dependence analysis for managing parallelism: Grouping

- **Background**: Tasks and Data decomposition has been done.
- All the identified tasks may not run in parallel.
- **Q**: How should related tasks be grouped to help manage the dependencies?
- Dependent, related tasks should be (uniquely?) grouped together.
  - Temporal dependency: If task A depends on the result of task *B*, then *A* must wait for the results from *B*. Q: Does *A* have to wait for *B* to terminate?
  - Concurrent dependency: Tasks are expected to run in parallel, and one depends on the updates of the other.
  - Independent tasks: Can run in parallel or in sequence. Is it always better to run them in parallel?
- Advantage of grouping.
  - Grouping enforces partial orders between tasks.
  - Application developer thinks of groups, instead of individual tasks.
- Example: Computing of individual rows.

Dependence analysis for managing parallelism: Ordering

- **Background**: Tasks and Data decomposition has been done. Dependent tasks have been grouped together.
- Ordering of the tasks and groups not trivial.
- **Q**: How should the groups be ordered to satisfy the constraints among the groups and in turn tasks?
- Dependent groups+tasks should be ordered to preserve the original semantics.
  - Should not be overly restrictive.
  - Ordering is imposed by: Data + Control dependencies.
  - Ordering can also be imposed by external factors: network, i/o and so on.
  - Ordering of independent tasks?
- Importance of grouping.
  - Ensures the program semantics.
  - A key step in program design.

Dependence analysis for managing parallelism: data sharing

Background: Tasks and Data decomposition has been done. Dependent tasks have been grouped together. The ordering between the groups and tasks have been identified.

- Groups and tasks have some level of dependency among each other.
- **Q**: How is data shared among the tasks?
- Identify the data updated/needed by individual tasks - task local data.
- Some data may be updated by multiple tasks - global data.
- Some data may be updated by one data used by multiple tasks - remote data

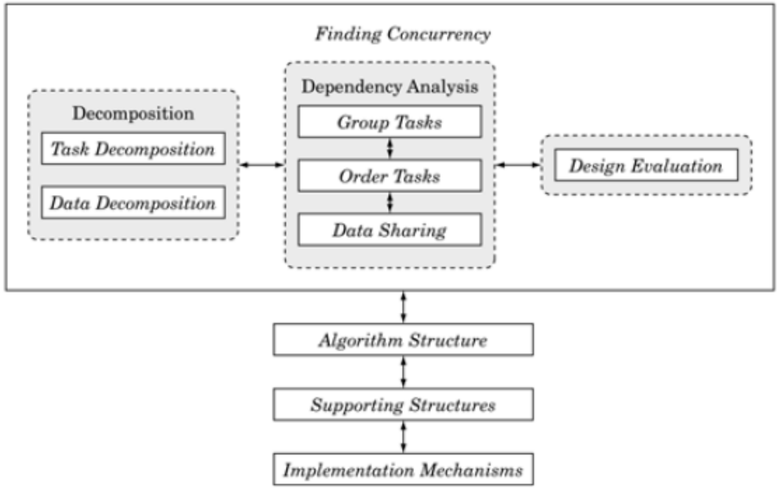
Issues in data sharing

- Identify the data being shared - directly follows from the decomposition.
- If sharing is done incorrectly - a task may get invalid data due to race condition.
- A naive way to guarantee correct shared data: synchronize every read with barriers.
- Synchronization of data across different tasks - may require communication. Options:
  - Overlap of communication and computation.
  - Privatization.
  - keep local copies of shared data.

One special case of sharing

- Accumulation/Reduction: Data being used to accumulate a result; sum, minimum, maximum, variance etc.
  - Each core has a separate copy of data,
  - accumulation happens in these local copies.
  - sub-results are further used to compute the final result.
- Example: Sum elements in an array A[1024]
  - Decompose the array into 32 chunk.
  - Accumulate each chunk separately.
  - Accumulate the sub results into the global “sum”.

Finding concurrency in a given problem - deep dive



Managing parallelism - design evaluation

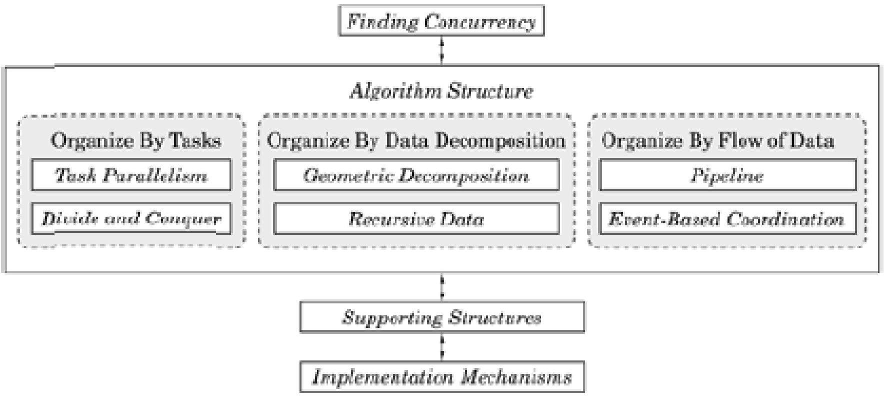
Background: Tasks and Data decomposition has been done. Dependent tasks have been grouped together. The ordering between the groups and tasks have been identified. A scheme for data sharing has also been identified.

- Of the multiple choices present at different points, we have chosen one.
- **Q**: Is the chosen path a “good” one?

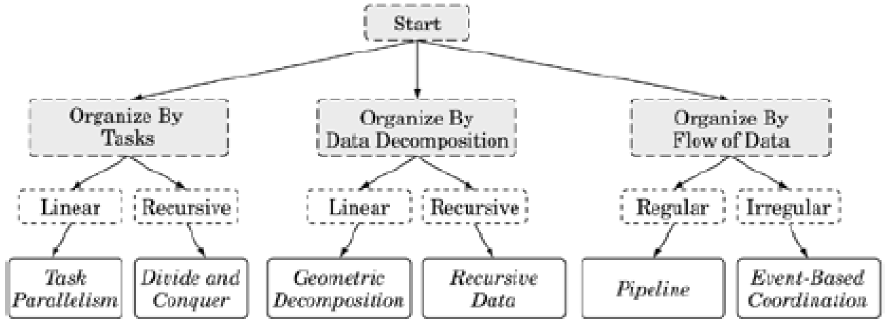
Design evaluation factors

- Suitability to the target platform (at a high level)
  - Number of cores / HW threads - too few/many tasks?
  - Homogeneous/Heterogeneous multi-cores? And work distribution.
  - Data distribution among the cores - equal/unequal?
  - Cost of communication - fine/coarse grained data sharing.
  - Amount of sharing - shared memory or distributed memory.
- Metrics: simplicity (qualitative) , Efficiency , Flexibility
- Flexibility
  - Flexible/Parametric over the number of cores/threads?
  - Flexible/Parametric over the number and size of data chunks?
  - Does it handle boundary cases?
- Efficiency.
  - Even load balancing?
  - Minimum overhead? - task creation, synchronization, communication.

Algorithm Structure - deep dive



Algorithm Structure design



Task Parallelism

Q: A problem is best decomposed into a collection of tasks that can execute concurrently. How to exploit the concurrency efficiently?

- Problem can be decomposed into a collection of concurrent tasks.
- Tasks can be completely independent or can have dependencies.
- Tasks can be known from the beginning (producer/consumer), tasks are created dynamically.
- Solution may or not require all the tasks to finish.

Challenges:

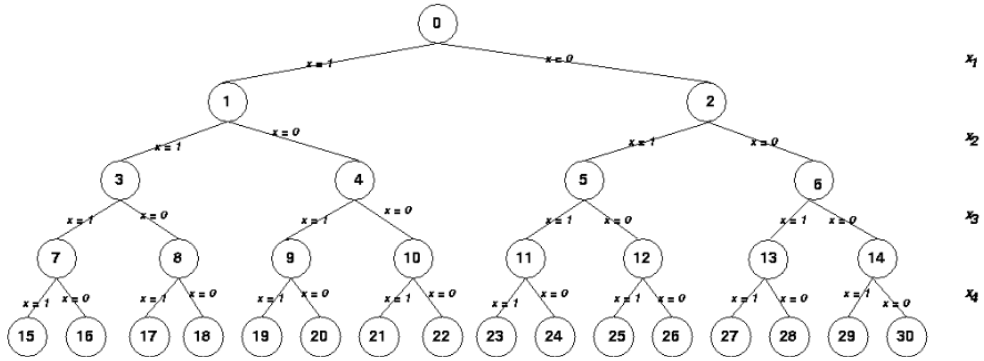
- Assign tasks to cores - to result in a simple, flexible and efficient execution.
- Address the dependencies correctly.

Factors in efficient Task parallel algorithm design

- Tasks:
  1. Enough Tasks to keep the cores busy.
  2. Advantage of creating the tasks should offset the overhead of creating and managing them.
- Dependencies
  1. Ordering constraints.
  2. Dependencies from shared data: synchronization, private data.
  3. Schedule: creation and scheduling.
- Schedule
  1. How are the tasks assigned to cores.
  2. How are the tasks scheduled.

Example: Task parallel algorithm

Machine	Job1	Job2	Job3	Job4
M1	4	4	3	5
M2	2	3	4	4

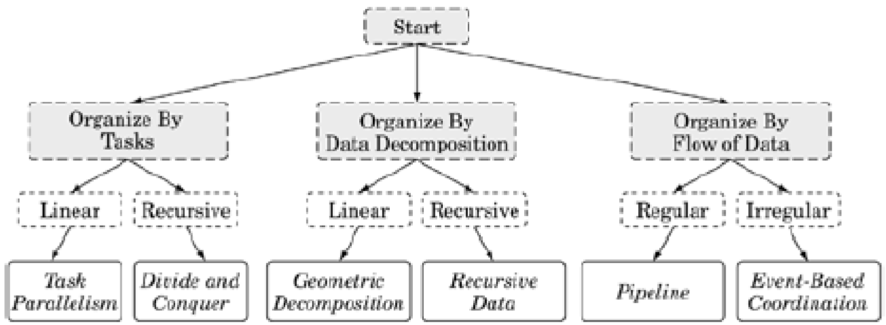


- Say Job1 to M1, Job2 to M2, Job3 to M1, Job4 to M2 = 7.

Solution to Branch and Bound ILP

- Maintain a list of tasks.
- Remove a solution from the list.
- **Examine the solution. Either discard it or declare it a solution, or add a sub-problem to task list.**
- The tasks depend depend on each other through the task-list.

Algorithm Structure design

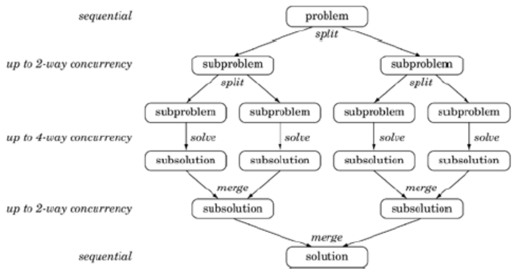


Divide and conquer

Q: Tasks are created recursively to solve a problem in a divide conquer strategy. How to exploit the concurrency?

- Divide and Conquer: Problem is solved by splitting it into a number of smaller subproblems. Examples?
- Each subproblems can be solved “fairly” independently. Directly or further divide and conquer.
- Solutions of the smaller problems is merged to compute the final solution.
- Each divide doubles the concurrency.
- Each merge halves the concurrency.

Divide and Conquer pattern: features



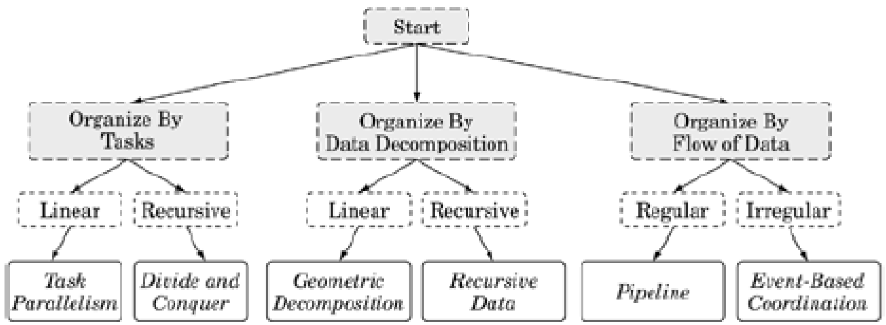
- The amount of exploitable concurrency varies.
- At the beginning and end very little exploitable concurrency.
- Note: “split” and “merge” are serial parts.
- Amdahl’s law - speed up constrained by the serial part. Impact?
- Too many parallel threads?
- What if cores are distributed? - data movement?
- Tasks are created dynamically - load balancing?
- What if the sub-problems are not equal-sized?

Divide and conquer - example Mergesort

```
int[] mergesort(int[]A,int L,int H){
   if (H - L <= 1) return;
   if (H-L <= T) {quickSort(A, L, H); return;}
   int m = (L+H)/2;
   A1 = mergesort(A, L, m);
   A2 = mergesort(A, m+1, H);
   return merge(A1, A2);
   // returns a merged sorted array.
}
```

- split cost?
- merge cost?
- Value of threshold T?

Algorithm Structure design



Geometric decomposition

Q: How can an algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”?

- Similar to decomposing a geometric region into subregions.
- Linear Data structures (such as arrays) - can be often decomposed into contiguous sub-structures.
- These individual tasks are processed in different concurrent tasks.
- Note: Sometimes all the required data for a task is present “locally” (embarrassingly parallel - Task parallelism pattern). And sometimes share data with “neighboring” chunks.

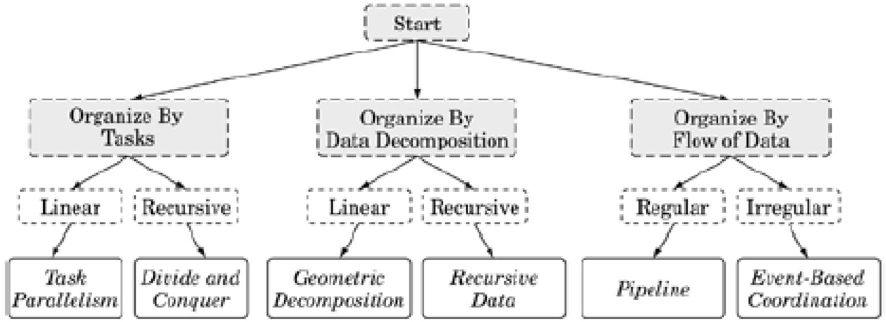
Challenges

- Ensure that each task has access to all data it needs.
- Mapping of chunks to cores giving good performance. Q: Why is it a challenge?
- Granularity of decomposition (coarse or fine-grain) - effect on efficiency? Parametric? Tweaked at compile time or runtime?
- Shape of the chunk: Regular/irregular?

Geometric decomposition: Matrix multiplication

$$
\begin{aligned}
C &= A \times B \\
&= \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \\
&= \begin{pmatrix} A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{pmatrix}
\end{aligned}
$$

Algorithm Structure design

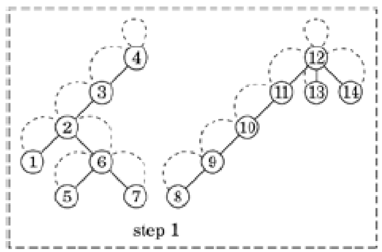


Recursive Data Pattern

Q: How can recursive data structures be partitioned so as that operations on them are performed in parallel?

- Linked list, tree, graphs . . .
- Inherently operations on recursive data structures are serial - as one has to sequentially move through the data structure.
- For example linked list traversal or traversing a binary tree.
- Sometimes it is possible to reshape operations to derive and exploit concurrency.

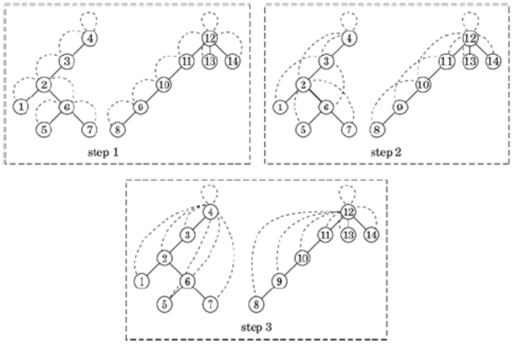
Recursive Data Pattern - example Find roots



- Given a forest of rooted trees: compute the root of each node.
- Serial version: Do a depth-first or breadth first traversal from root to the leaf nodes.
- For each visited node - set the root. Total running time?

Q: Is there concurrency?

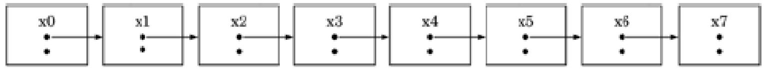
Recursive Data structures: Parallel find roots



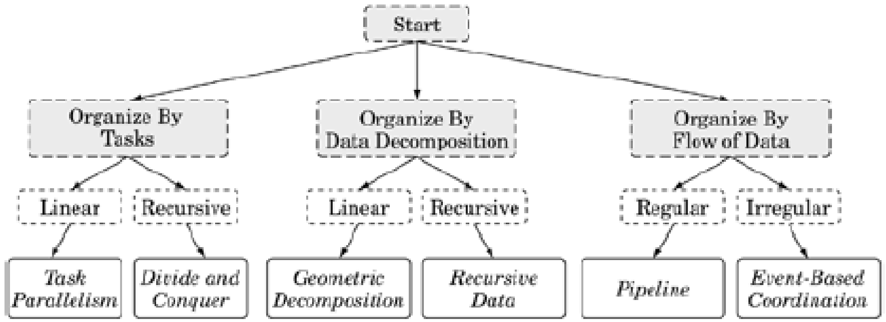
- Transformed the original serial computation to one where we compute partial result and repeatedly combine partial results. Total Cost = ?
- Total cost = $O(N \log N)$
- However, if we exploit the parallelism - running time will come down to $O(\log N)$.

Parallelizing recursive data structures

- Recasting the problem increases the cost. Find a way to get it back.
- Effective exploitation of the derived concurrency depends on factors such as - amount of work available for each task, amount of serial code . . .
- Restructuring may make the solution complex.
- Requirement of synchronization - Why?
- Another example: Find partial sums in a linked list.



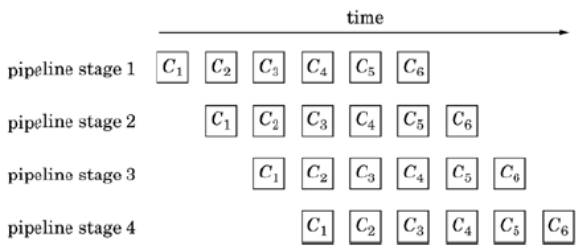
Algorithm Structure design



Pipieline pattern

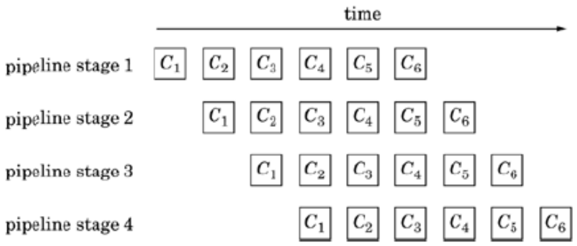
Q: The computation may involve performing similar sets of operations on many sets of data. Is there concurrency? How to exploit it?

- Factory assembly line, Network Packet processing, Instruction processing in CPUs etc.



- There are ordering constraints on each operation on any one set of data: Operation C_2 can be undertaken only after C_1.
- Key requirement: Number of operations > 1.

Pipeline pattern features

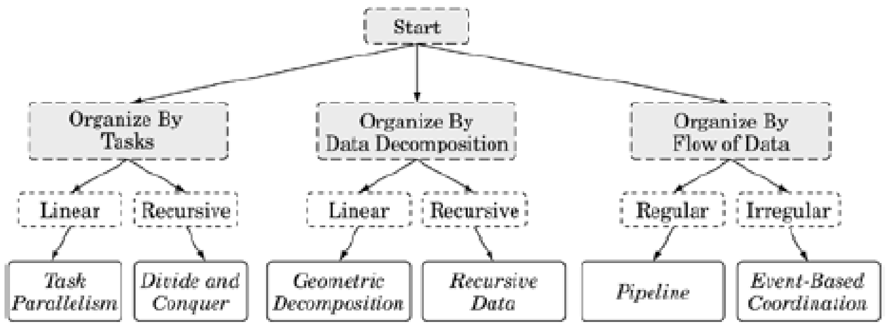


- Once the pipeline is full maximum parallelism is observed.
- Number of stages should be small compared to the number of items processed.
- Efficiency improves if time taken in each stage is roughly the same. Else?
- Amount of concurrency depends on the number of stages.
- Too many stages, disadvantage?
- Communication across stages?

Pipeline pattern. Issues

- Error handling.
  - Create a separate task for error handling - which will run exception routines.
- Processor allocation, load balancing
- Throughput and Latency.

Overall big picture



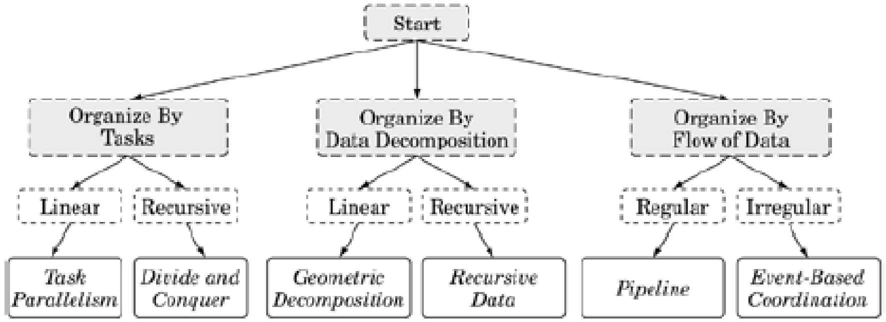
Event based coordination

Challenges

- Identifying the tasks.
- Identifying the events flow.
- Enforcing the events ordering.
- Avoiding deadlock.
- Efficient communication of events.

Left for self reading.

Algorithm Structure design



Supporting structures

- We have identified concurrency, and established an algorithm structure.
- Now how to implement the algorithm?

Issues

- Clarity of abstraction - from algorithm to source code.
- Scalability - how many processors can it use?
- Efficiency - utilizing the resource of the computer, *efficiently*. Example?
- Maintainability - is it easy to debug, verify and modify?
- Environment - hardware and programming environment.

SPMD pattern

- Each UE executes the same program, but has different data.
- They can follow different paths through the program. How?
- Code at different UEs can differentiate with each other using a unique ID.
- Assumes that each underlying hardware are similar.

Challenges

- Interactions among the seemingly independent activities of UEs.
- Clarity, Scalability, Efficiency, Maintainability (1m cores), Environment.
- How to handle code like initialization, finalization etc?

SPMD example

$$\pi = \int_0^1 \frac{4}{1+x^x} dx$$

```
int main () {
  // Initialization start
  int i;
  int numSteps = 1000000;
  double x, pi, step, sum = 0.0;
  step = 1.0/(double) numSteps;
  // Initialization end
  for (i=0;i< numSteps; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x); }
  // Finalization start
  pi = step * sum;
  printf("pi %lf\n",pi);
  return 0;
  // Finalization end
}
```

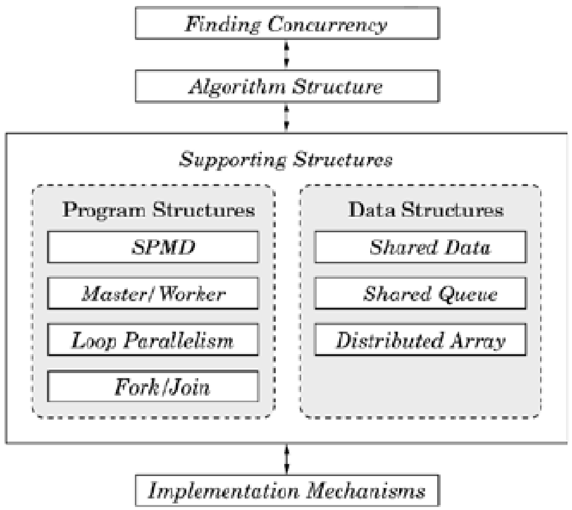
SPMD translation. Inefficient?

```
int main () {

  int i;
  int numSteps = 1000000;
  double x, pi, step, sum = 0.0;
  step = 1.0/(double) numSteps;
  int numProcs = numSteps;
  int myID = getMyId();

  i = myID;
  x = (i+0.5)*step;
  sum = sum + 4.0/(1.0+x*x);

  sum = step * sum;
  DoReductionOverAllProcs(&sum, &pi); // blocking.
  if (myID == 0) printf("pi %lf\n",pi);
  return 0;

}
```


SPMD translation. Better?

```
int main () {
  int i; int numSteps = 1000000;
  double x, pi, step, sum = 0.0;
  step = 1.0/(double) numSteps;
  int numProcs = getNumProcs();
  int myID = getMyId();
  step = 1.0/numSteps;

  iStart = myID * (numSteps / numprocs);
  iEnd = iStart * (numSteps / numprocs);
  if (myID == numProcs-1) iEnd = numSteps;

  for (i = iStart; i < iEnd; ++i){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x); }
  sum = step * sum;
  DoReductionOverAllProcs(&sum, &pi); // blocking.
  if (myID == 0) printf("pi %lf\n",pi);
  return 0; }
```


Supporting structure



Master/Worker

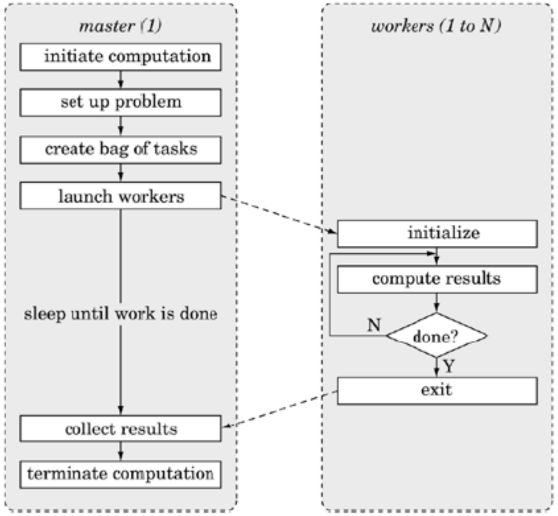
Situation

- workload at each task is variable and unpredictable (what if predictable?).
- Not easy to map to loop-based computation.
- The underlying hardware have different capacities.

Master/Worker pattern

- Has a logical master, and one or more instances of workers.
- Computation by each worker may vary.
- The master starts computation and creates a set of tasks.
- Master waits for tasks to get over.

Master/Worker layout



Q: How to implement the set of tasks? Characteristics of this data structure?

Master/Worker Issues

- Has good scalability, if number of tasks greatly exceed the number of workers, and each worker roughly gets the same amount of work (Why?).
- Size of tasks should not be too small. Why?
- Can work with any hardware platform.
- How to detect completion? When can the workers not wait but shutdown?
  - Easy if all tasks are ready before workers start.
  - Use of a poison-pill in the work-queue.
  - What if the workers can also add tasks? Issues?
  - Issues with asynchronous message passing systems?
  - How to handle fault tolerance? - did the task finish?

Variations

- Master can also become a worker.
- Distributed task queue instead of a centralized task queue. (dis)advantages?

Master/Worker template for master

```
int nTasks // Number of tasks
int nWorkers // Number of workers
public static SharedQueue taskQueue; // global task queue
public static SharedQueue resultsQueue; // queue to hold result:
void master() {
// Create and initialize shared data structures
taskQueue = new SharedQueue();
globalResults = new SharedQueue();
for (int i = 0; i < nTasks; i++)
   enqueue(taskQueue, i);

// Create nWorkers threads
ForkJoin (nWorkers);

consumeResults (nTasks);
}
```


Master/Worker - ForkJoin

```
void ForkJoin(int nWorker){
  Thread [] t = new Threads[nWorkers];
  for (int i=0;i<nWorker;++i) {
    t[i] = new Thread(new Worker()) }
  for (int i=0;i<nWorker;++i) {
    t[i].join();}
}
```

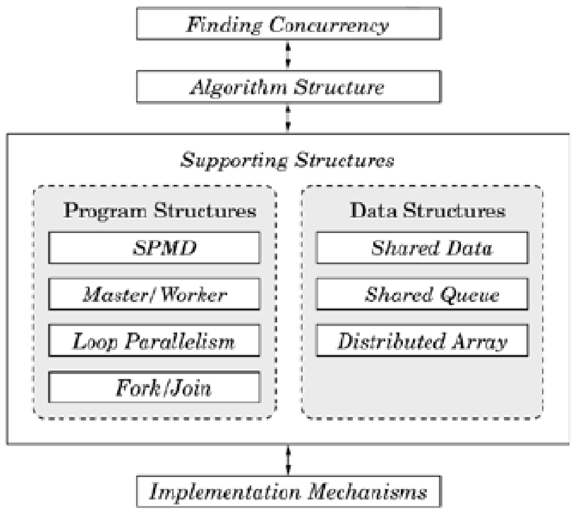
Master/Worker - template for worker

```
class Worker(){
  public void run() {
    while (!(Master.taskQueue.empty()){
      // atomically dequeue.
      // do computation.
      // add to globalResults atomically
    } } }
```

Known uses

- SETI@HOME
- Map Reduce
  - "Map" step: The master node takes the input, partitions it up into smaller sub-problems, and distributes those to worker nodes.
    - A worker may again partition the problem – multi-level tree structure.
    - The worker node processes that smaller problem, and passes the answer back to its master node.
  - "Reduce" step: Master node takes all the answers and combines them to get the output the answer to the original problem.

Supporting structure



Loop Parallelization

- A program has many computationally intensive loops, with "independent" iterations.
- Goal: Parallelize the loops and get most of the benefits.
- Very narrow focus.
- Typical application: scientific and high performance computation.
- Impact of Amdahl's law?
- Quite amenable to refactoring type of incremental parallelization. Advantage?
- Impact on distributed memory systems?
- Good if computation done in iterations compensates the cost of thread creation - how to improve the tradeoff? Coalescing, merging.

Loop coalescing and merging for parallelization

Merging/Fusion

```
for (i : 1..n) {
  S1
}
for (j : 1..n) {
  S2
}
-->
for (i : 1..n) {
  S1
  j = i;
  S2
}
```

Coalescing

```
for (i : 0..m) {
  for (j : 0..n) {
    S
}
-->
for (ij : 0..m*n) {
  j = ij % n;
  i = ij / n;
  S
}
```

Loop parallelization issues

- Distributed memory architectures.
- False sharing : variables not needed to be shared, but are in the same same cache line. Can incur high overheads.
- Seen in systems with distributed, coherent caches.
- The caching protocol may force the reload of a cache line despite a lack of logical necessity.

```
foreach(j : [0..N]) {
  for(i=0; i<M; i++){
    A[j]+= compute(j,i);
  }
}
```

```
foreach(j : [0..N]) {
  double tmp;
  for(i=0; i<M; i++){
    tmp += compute(j,i);
  }
  atomic A[j] += tmp;
}
```


Loop parallelization example

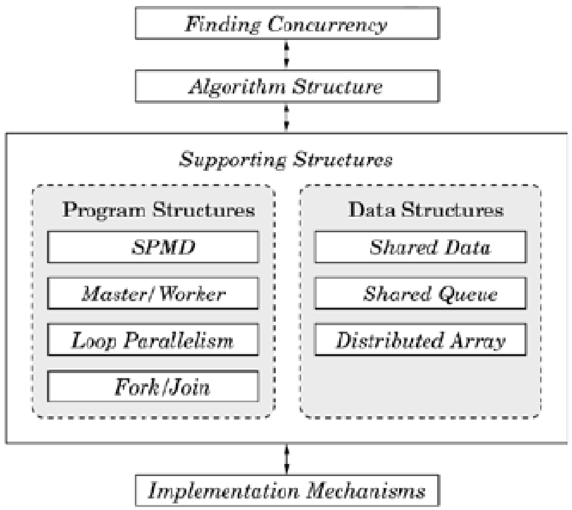
$$\pi = \int_0^1 \frac{4}{1+x^x} dx$$

```
int main () {

  int i,numSteps = 1000000;
  double x,pi,step,sum=0.0;
  step=1.0/(double)numSteps;

  for(i: [0..numSteps]){
    x=(i+0.5)*step;
    sum=sum+4.0/(1.0+x*x);}

  pi=step*sum;
  printf("pi %lf\n",pi);
  return 0; }
```

```
int main () {

  int i,numSteps=1000000;
  double pi,step,sum=0.0;
  step=1.0/(double)numSteps;

  forall(i: [0..numSteps]){
    double x=(i+0.5)*step;
    double tmp=4.0/(1.0+x*x)
    atomic sum=sum+tmp; }

  pi = step * sum;
  printf("pi %lf\n",pi);
  return 0; }
```

Reading material: Automatic loop parallelization.

Supporting structure



Fork/Join

- The number of concurrent tasks varies as the program executes.
- Parallelism beyond just loops.
- Tasks created dynamically (beyond master-worker).
- One or more tasks waits for the created tasks to terminate.
- Each task may or not result in an actual UE. Many-to-one mapping. Examples?

Fork/Join - example Mergesort

```
int[] mergesort(int[]A,int L,int H){
   if (H-L <= T) {quickSort(A, L, H); return;}
   int m = (L+H)/2;
   A1 = mergesort(A, L, m); // fork
   A2 = mergesort(A, m+1, H); // fork
   // join.
   return merge(A1, A2);
   // returns a merged sorted array.
}
```

Issues

- Cost.
- Alternatives?

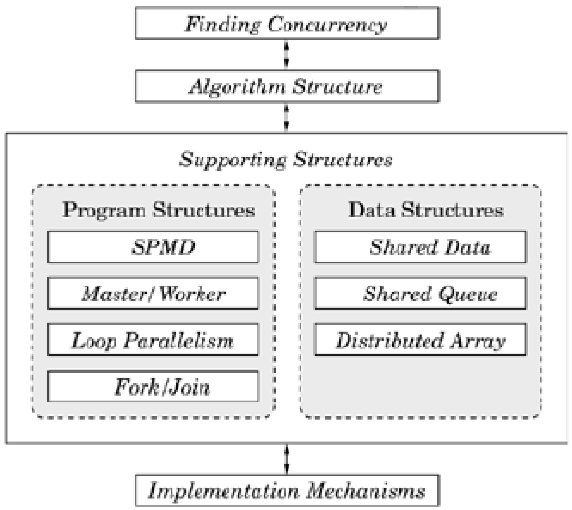
Supporting structures and algorithm structure

	Task Parallelism	Divide and Conquer	Geometric Decomposition	Recursive Data	Pipeline	Event-Based Coordination
SPMD	★★★★	★★★	★★★★	★★	★★★	★★
Loop Parallelism	★★★★	★★	★★★			
Master/Worker	★★★★	★★	★	★	★	★
Fork/Join	★★	★★★★	★★		★★★★	★★★★

Homework

	OpenMP	MPI	Java	X10	UPC	Cilk	Hadoop
SPMD	★★★	★★★★	★★				
Loop Parallelism	★★★★	★	★★★				
Master/Worker	★★	★★★	★★★				
Fork/Join	★★★		★★★★				

Supporting structure



Shared Data

Million dollar question: How to handle shared data?

- Managing shared data incurs overhead.
- Scalability can become an issue.
- Can lead to programmability issues.
- Avoid if possible - by
  - replication,
  - privatization,
  - reduction.
- Use appropriate concurrency control. Why?
  - Should preserve the semantics.
  - Should not be too conservative.
- Shared data organization: distributed or at a central location?
- Shared Queue (remember master-worker?) is a type of shared data.

Issues with shared data

- Data race and interference: Two shared activities access a shared data. And at least one of them is a write. The activities said to interfere.

```
forall (i:[1..n]) {
  sum += A[i];
}
```

```
for (i[1..n]) {
 forall (j=1;j<m;++j) {
   A[i][j]=(A[i-1][j-1]+A[i-1][j]+A[i-1][j+1])/3;
 }
}
```

- Dependencies : Use synchronization (locks, barriers, atomics, . . .) to enforce the dependencies.
  - How to implement all-to-all synchronization?

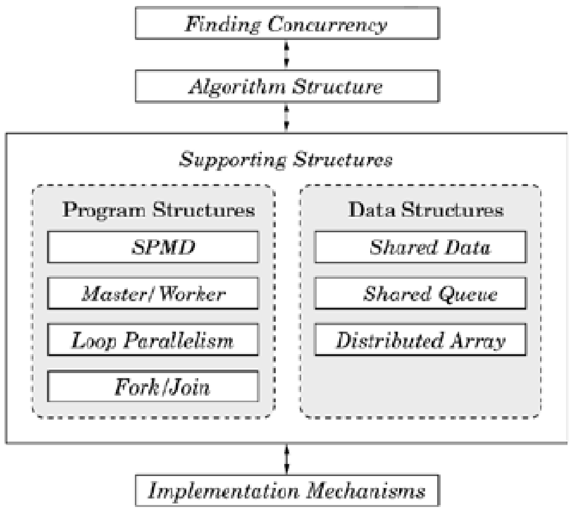
Issues with shared data

- Deadlocks : two or more competing actions are each waiting for the other to finish.

  (Example via nested locks) $\frac{\text{lockA} \rightarrow \text{lockB}}{\text{lockB} \rightarrow \text{lockA}}$

  One way to avoid: partial order among locks. Locks are acquired in an order respecting the partial order.
- Livelocks : the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Example: recovery from deadlock - If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered leading to a livelock
- Locality : Trivial if data is not shared.
- Memory synchronization: when memory / cache is distributed.
- Task scheduling - tasks might be suspended for access to shared data. Minimize the wait.

Supporting structure

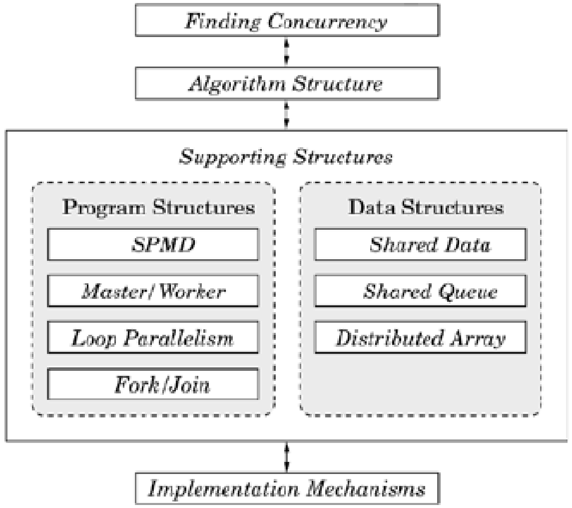


Distributed Array

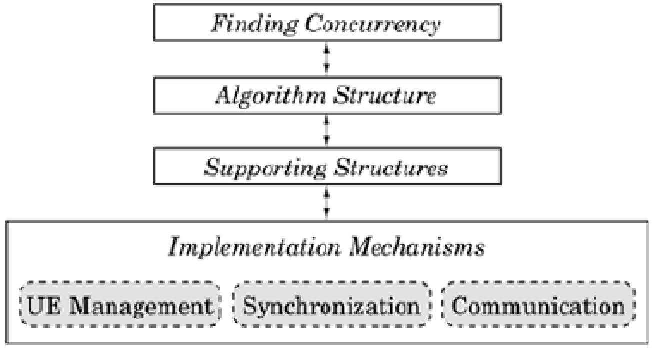
Arrays often are partitioned between multiple tasks.
Goal: Efficient code, programmability.

- Distribute the arrays such that elelement needed by a task is “available” and “nearby”.
- Array element redistribution?
- An abstraction is needed: a map from elements to places.
- Some standard ones: Blocked, Cyclic, Blocked cyclic, Unique,
- Chosing a distribution.

Supporting structure



Implementation Mechanisms



UE management

- UE - unit of execution (a process / thread / activity)
- Difference between process / thread / activity.
- Management = Creation, execution, termination.
- Varies with different underlying languages.
- Go back to first few lectures for a recap.

Synchronization: Memory synchronization and fences

Synchronization: Enforces constraint among parallel events.

-

```
done=true;                          done = false;
while(done) ;
```

  - Value may be present in cache. cache coherence may take care.
  - Value may be present in a register - Culprit compiler.
  - Value may not be read. How?
-

x = y = 0

Thread 1	Thread 2
1: r1 = x	4: x = 1
2: y = 1	r3 = y
3: r2 = x	

r1 == r2 == r3 == 0. Possible?

Synchronization: Memory synchronization and fences

- A memory fences guarantees that the UEs will see a consistent view of memory.
- Writes performed before the fence will be visible to reads performed after the fence.
- Reads performed after the fence will obtain a value written no earlier than the latest write before the fence.
- Only for shared memory.
- Explicit management can be error prone. High level: OpenMP flush, shared, Java - volatile. *Read yourself.*

Syncrhonization: Barriers

Barrier is a synchronization point at which every member of a collection of UEs must arrive before any member can proceed.

- MPI_Barrier, join, finish, clocks, phasers
- Implemented underneath via passing messages.

Phasers[1]

- **Phaser allocation**
  - **Phaser ph = new Phaser(mode)**
    - Phaser **ph** is allocated with **registration mode**
    - Mode: **SINGLE**

```
            SINGLE
              |
      SIG_WAIT(default)
         /        \
    SIGNAL        WAIT
```

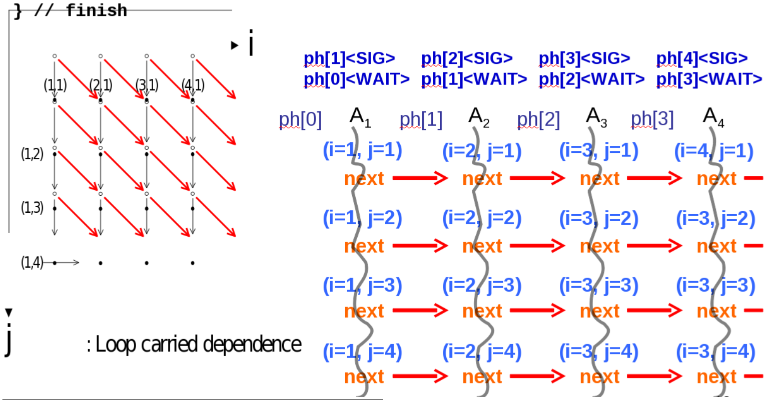
- Mode defines capability
- There is a lattice ordering of capabilities

- **Activity registration**
  - **async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, ...) {STMT}**
    - Spawned activity is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, ...
    - child activity's capabilities must be subset of parent's
- **Synchronization**
  - **next:**
    - Advance each phaser that activity is registered on to its next phase
    - Semantics depends on registration mode

[1] Thanks - Jun Shirako

Power of Phaser - pipeline parallelism[2]

```
finish {
  phaser [] ph = new phaser[m+1];
  foreach (point [i] : [1:m-1]) phased (ph[i]<SIG>, ph[i-1]<WAIT>)
    for (int j = 1; j < n; j++) {
      a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
      next;
    } // for
  } // foreach
} // finish
```

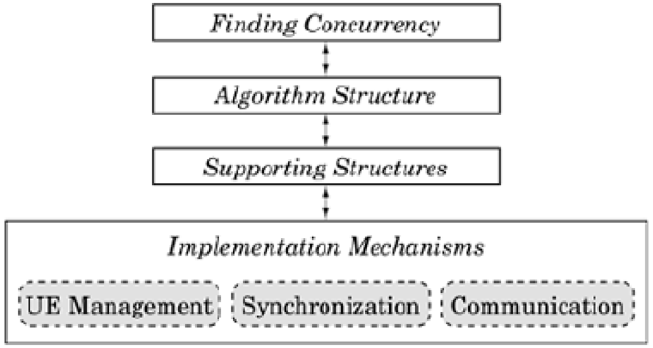


[2] Thanks - Jun Shirako

Syncrhonization

- Memory fence
- Barriers
- Mutual exclusion: Java `synchronized`, `omp_set_lock`, `omp_unset_lock`.

Implementation Mechanisms



Communication

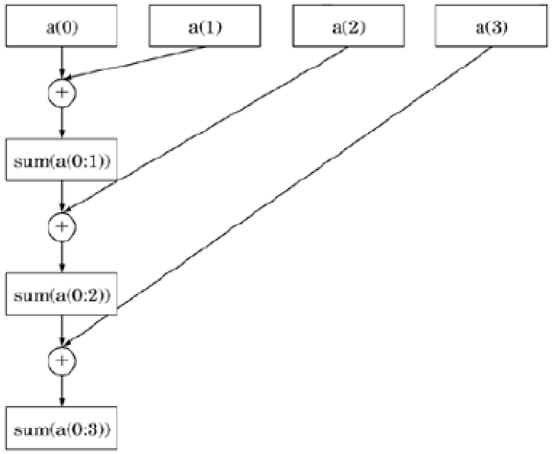
- UEs need to exchange information.
  - Shared memory - easy. Challenge - synchronize the memory access so that results are correct irrespective of scheduling.
  - distributed memory - not much need for synchronization to protect the resources. $\rightarrow$ Communication plays a big role.
- One to one communication :
- Between all UEs in one event: Collective communication.

Collective communication

When multiple UEs participate in a single communication event, the event is called a collective communication operation. Examples:

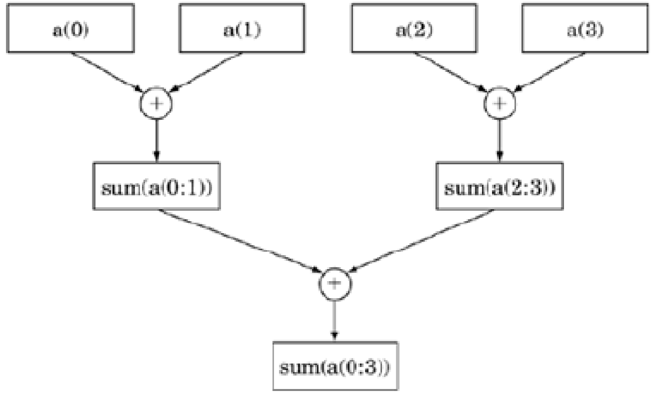
- Broadcast: a mechanism to send single message to all UEs.
- Barriers : a synchronization point.
- Reduction: Take a collection of objects, one from each UE, and "combine" into a single value;
  - combined value present only on one UE?
  - combined value present on all UEs?

Serial reduction



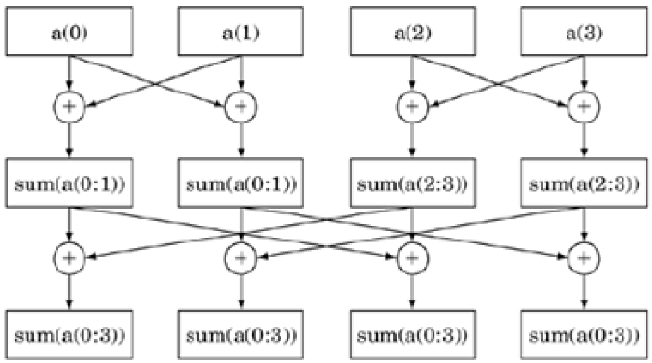
- Reduction with n items takes n steps.
- Useful especially if the reduction operator is not associative.
- Only one UE knows the result.

Tree based reduction



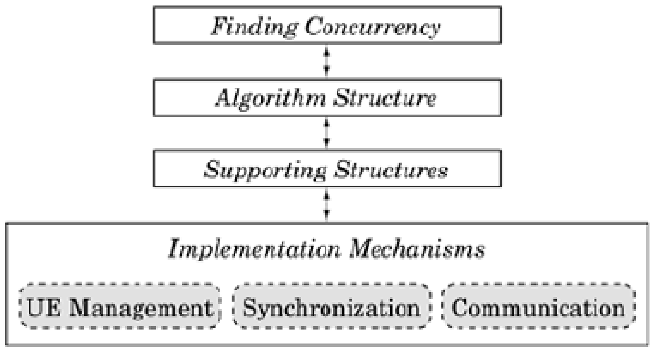
- Reduction with 2^n items takes n steps.
- What if number of UEs $<$ number of data items?
- Only one UE knows the result.
- Associative + Commutative or don't care (example?)

Recursive doubling

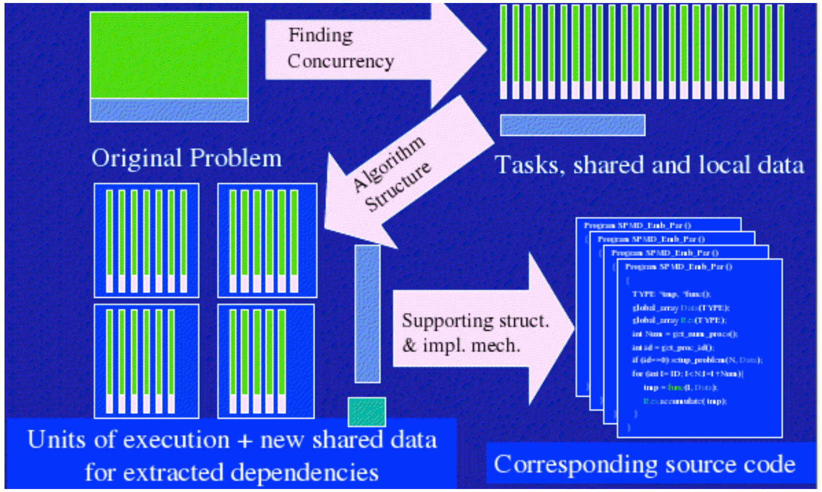


- Reduction with $2 \times n$ items takes n steps.
- What if number of UEs $<$ number of data items?
- All UEs know the result.

Implementation Mechanisms



Overall big picture



Sources

- Patterns for Parallel Programming: Sandors, Massingills.
- multicoreinfo.com
- Wikipedia
- fixstars.com
- Jernej Barbic slides.
- Loop Chunking in the presence of synchronization.
- Java Memory Model JSR-133: “Java Memory Model and Thread Specification Revision”