CS3400 - Principles of Software Engineering

Software Engineering for Multicore Systems

V. Krishna Nandivada

IIT Madras

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

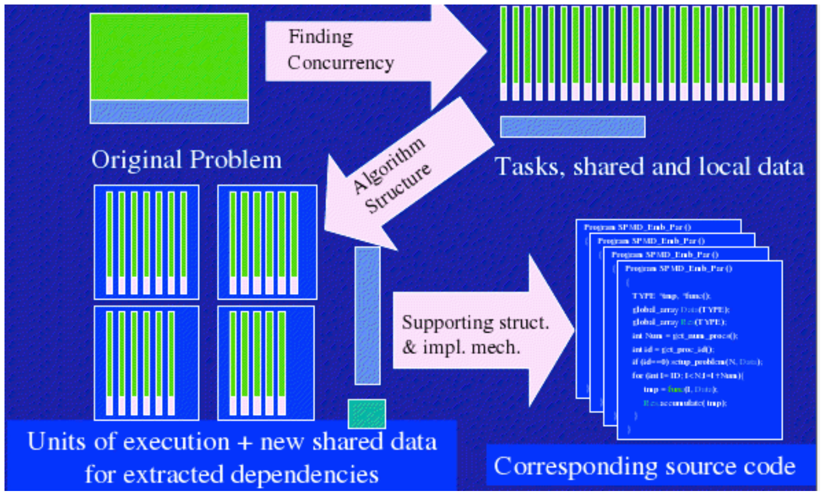
Helps algorithm to be implemented.

Implementation Mechanisms

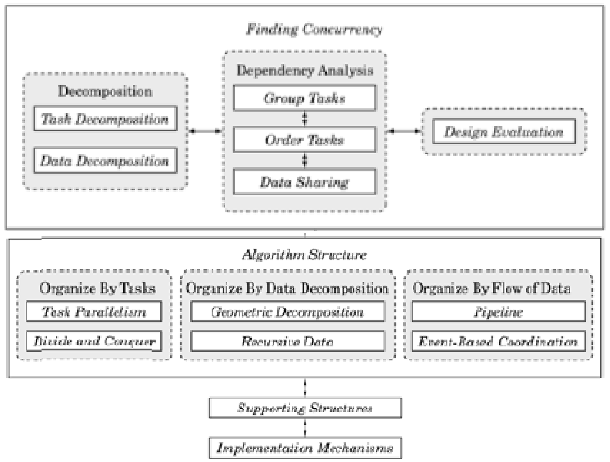
How the high level specifications are mapped.

Goal: Identify patterns in each stage.

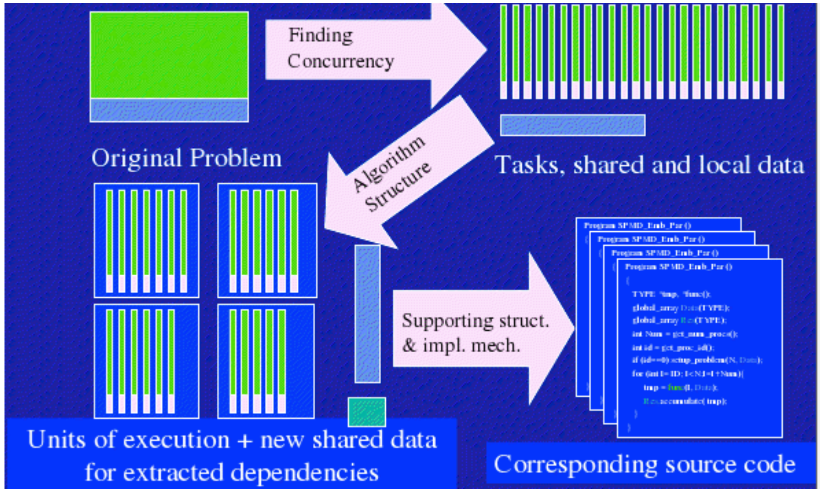
Overall big picture



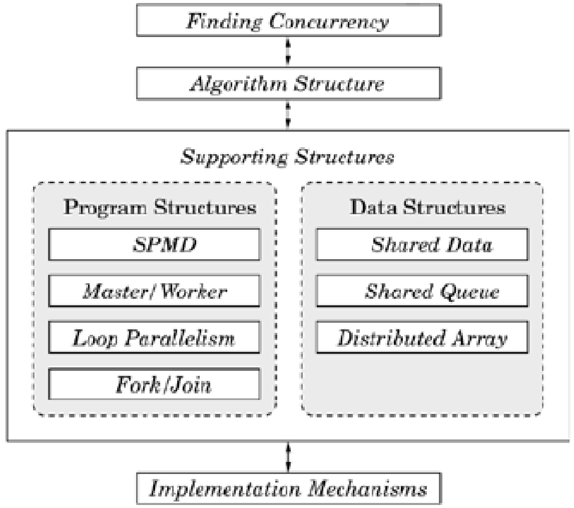
Finding concurrency and Algorithm Structure



Overall big picture



Supporting structure



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

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

```
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
}
```

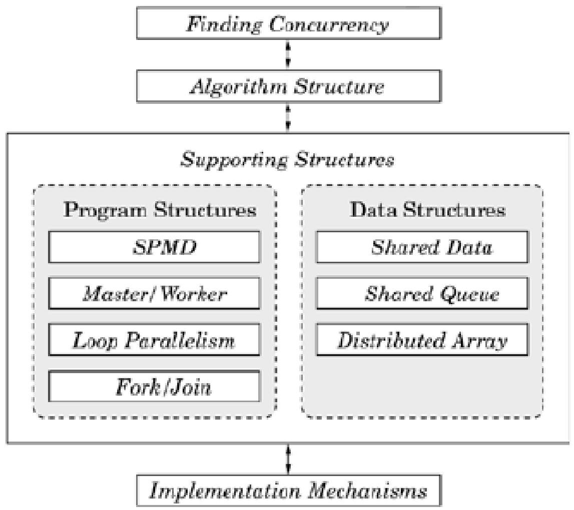

SPMD translation. Inefficient?

```
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

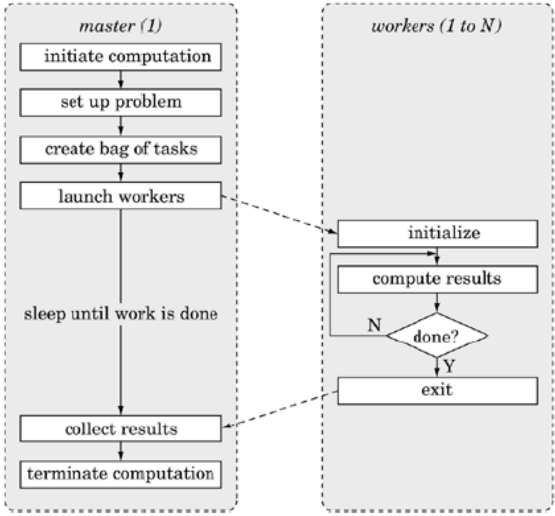
Situation

- 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 re
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();}
}
```

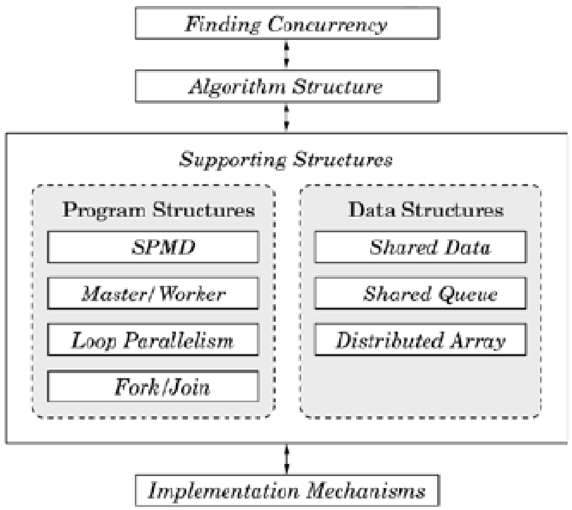

Master/Worker - template for worker

```
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

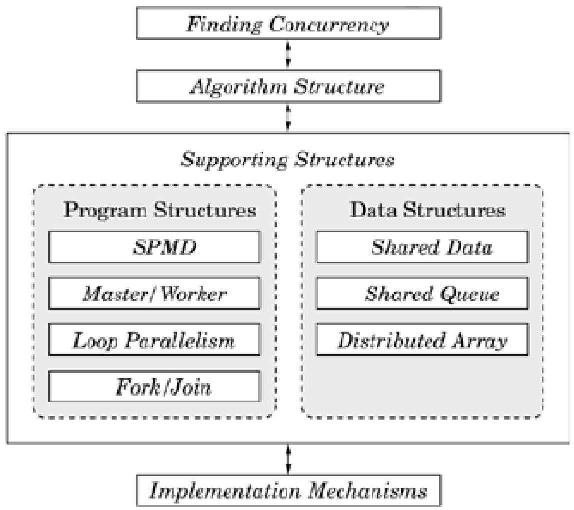
$$\pi = \int_0^1 \frac{4}{1+x^x} dx$$

```
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?

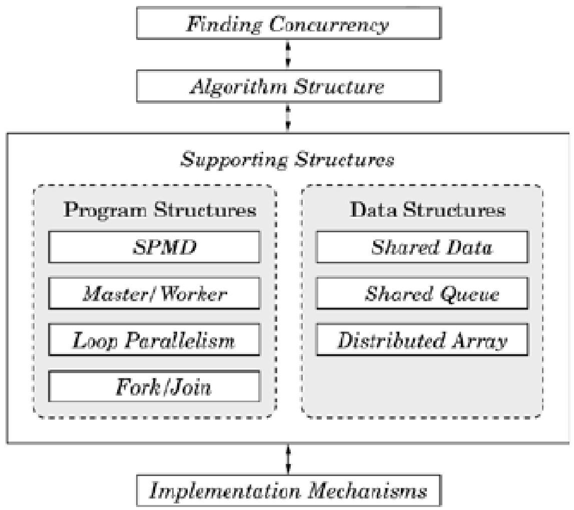
Supporting structures and algorithm structure

	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?

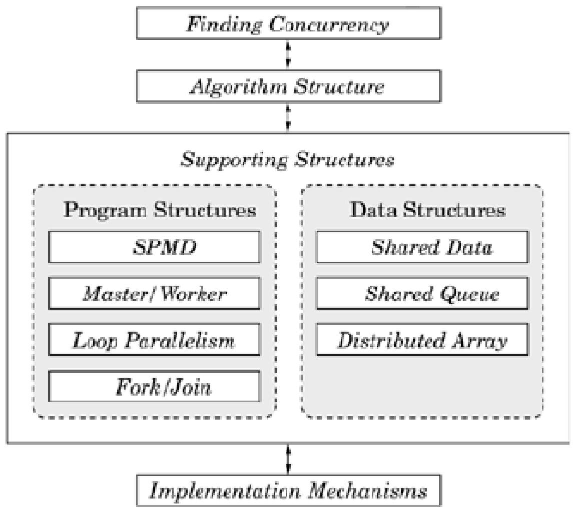
Issues with shared data

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

  (Example via nested locks) $\dfrac{\text{lockA} \to \text{lockB}}{\text{lockB} \to \text{lockA}}$

  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

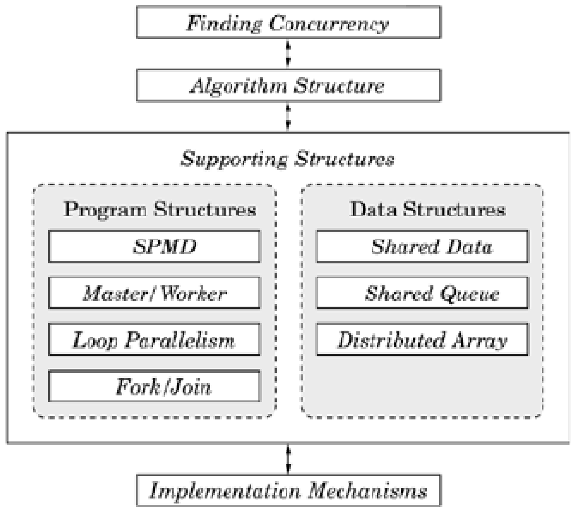


Distributed Array

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



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"