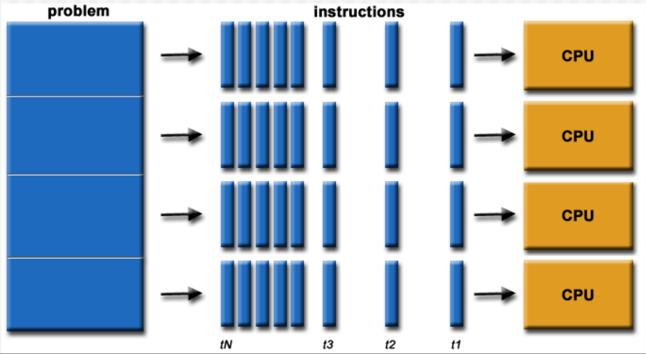
Lecture 4 & 5
Different Kinds of Parallelism

- Dependencies
- Instruction Level Parallelism
- Task Parallelism (L5)
- Data Parallelism (L5)

1

Recap: Parallel Computing

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions



2

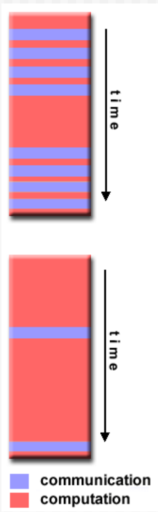
Terminology: Granularity

- **Computation / Communication Ratio:**
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.The granularity of parallelism is denoting the frequency of interactions among parallel activities

3

Granularity Cont’d

- **Fine-grain Parallelism:**
  - Relatively small amounts of computational work are done between communication events
  - Low computation to communication ratio
  - Facilitates load balancing
  - Implies high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.
- **Coarse-grain Parallelism:**
  - Relatively large amounts of computational work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently



4

Granularity Cont'd

- Task parallelism is typically coarse grained while data parallelism is often fine grained
- Exceptions: "embarrassingly parallel programs"
  - No or little need for synchronization/communication
  - E.g: search over large data sets (Seti@home, particle physics, …)

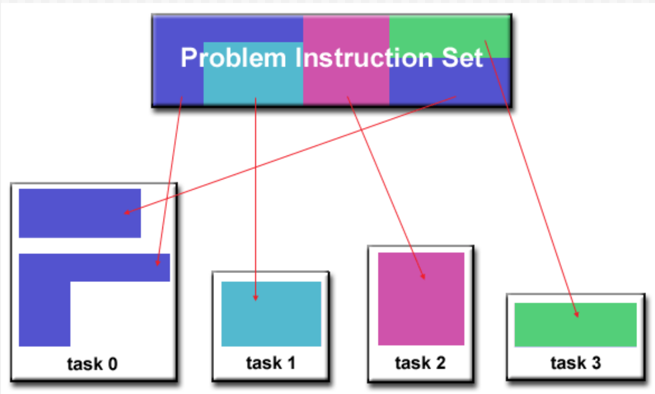
5

Task Parallelism

6

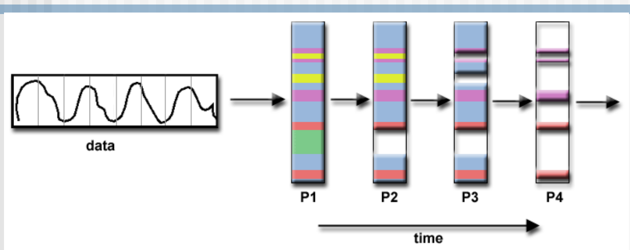
Task Parallelism

- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Also called "Functional Decomposition"

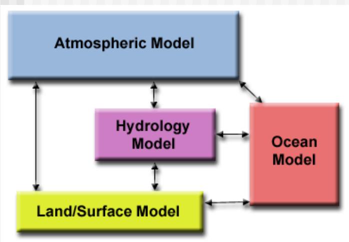


7

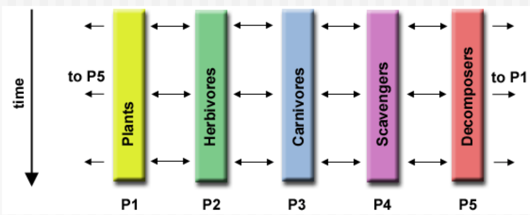
Task Parallelism Examples



Signal filtering



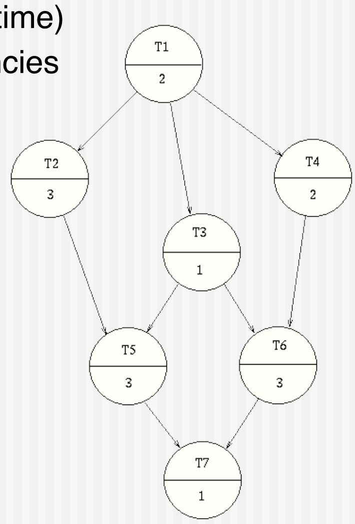
Climate modeling



Ecosystem modeling

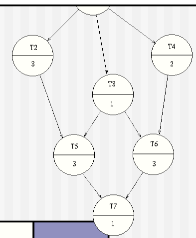
Task Graph

- Nodes denote tasks (and time)
- Vertices denote dependencies

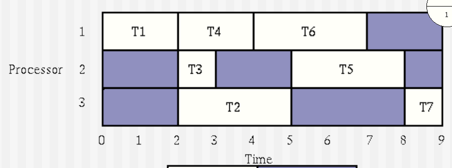


9

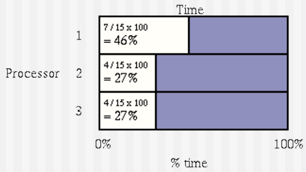
Task Graph Scheduling



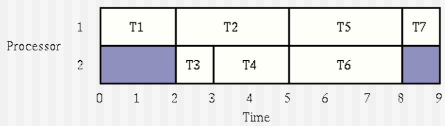
- Tasks can be scheduled freely as long as dependencies are honored
- Possible schedule



- Utilization



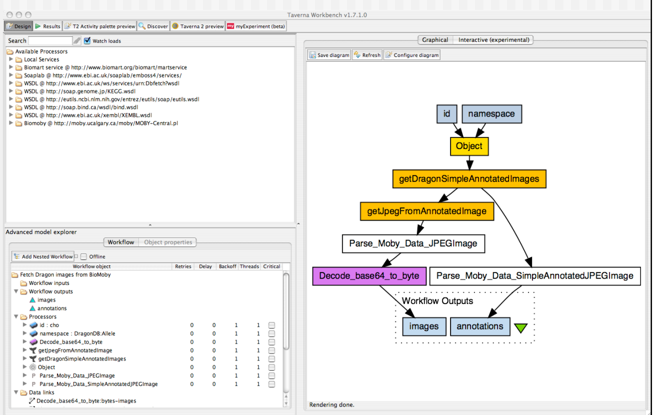
- 2 processors yield better utilization



10

Task Graph Scheduling

- Many tools out there support the construction and scheduling of task graphs
  - Issue: optimal graph decomposition is NP-hard
- Workflow systems typically also take different resource requirements into account
  - E.g. Taverna



Task Parallelism Summary

- Often pipelined approaches
- "Natural" approach to parallelism
- Typically good efficiency
  - Tasks proceed without interactions
  - Synchronization/communication needed at the end
- In practice scalability is limited
  - Problem can be split only into a finite set of different tasks

12

Task Parallelism Example

- Prepare a banquet

1. Appetizer
2. Salad
3. Main course
4. Dessert

- Degree of task parallelism limited to 4
- How can we increase the degree of parallelism?

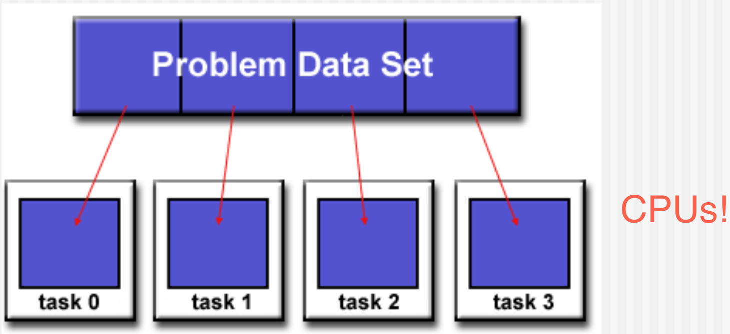
13

Data Parallelism

14

Data Parallelism

- The data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.
- Also called “Domain Decomposition”



15

Data Parallelism Example

- Prepare a banquet
- *N* meals are required
- Each of the *P* chefs prepares *N/P* meals
- Can be combined with task parallelism
  - P_1 chefs work on appetizer, P_2 chefs on salad, P_3 chefs on main course and P_4 chefs work on dessert

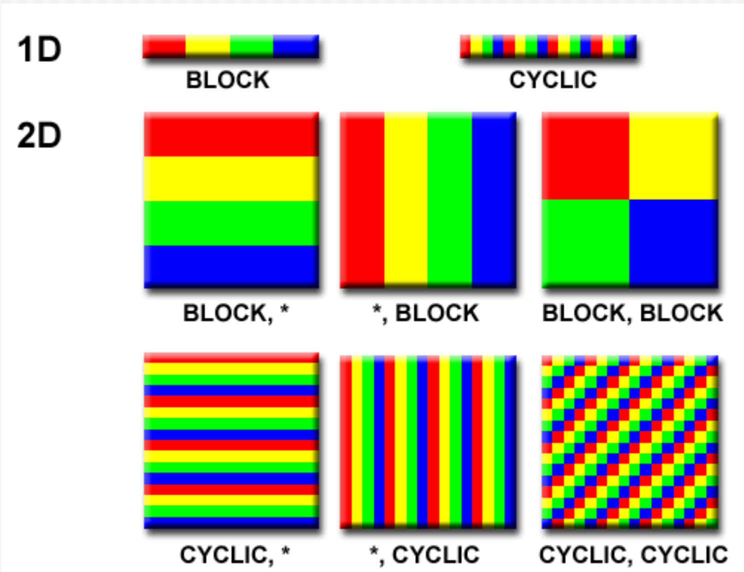
16

How to Partition Data

- Distribution Function:
  - *f(N)->P*; *N* denotes the data index and *P* the target processor
- Typical strategies are
  - Block
    - Distribute data in equal blocks over available processors
  - Cyclic
    - Distribute individual data items in round robin fashion over available processors
  - "*"
    - Replicate along a dimension
  - Irregular
    - Distribute data in over the processors using any kind of distribution function

17

Typical Data Distributions



18

Loop parallelization

- Data Parallelism is typically exploited by parallelizing loops
  - This is where most of the work of the program is typically done
- How to deal with dependencies?
  - Loop independent dependencies and
  - Loop carried dependencies
- How to distribute data?
  - Try avoid excessive communication
  - Analyze access patterns

19

Dependencies

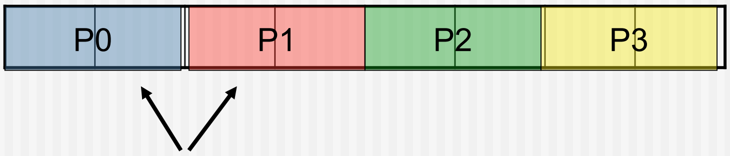
- Loop independent dependencies in loop body
  - Does prevent statement reordering and other ILP but does NOT prevent to parallelize the loop
- Loop dependent dependencies
  - Prevents parallelization of loop
- Special case: nested loops
  - Try loop interchange to move dependency inside so outer loop can be parallelized and independent workload is maximized — MIMD
  - Try to move dependency to outer loop so inner loop can be parallelized — SIMD

20

Access Patterns

- Stencils are a typical access pattern

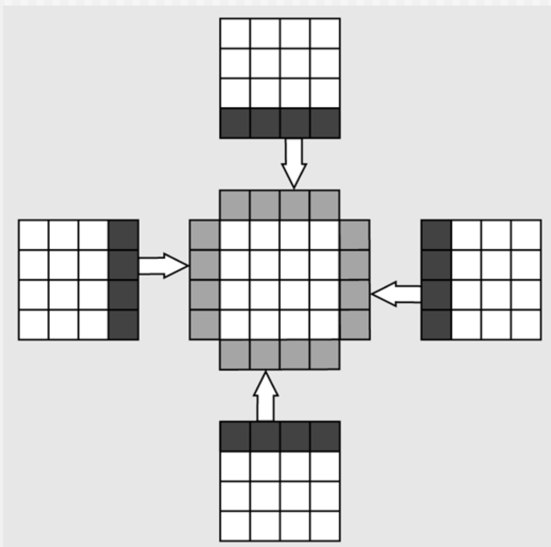
```
… = … a[i-1]+a[i]+a[i+1]
```



Overlap area

- Replicate overlap area or communicate it early on to avoid excessive communication inside loop

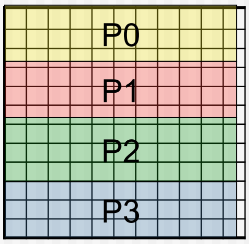
2D Overlap Area



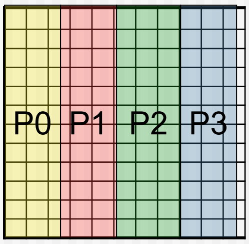
Access Patterns

- Matrix-Matrix Multiplication

```
do i=1,n
  do j=1,n
    do k=1,n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    end do
  end do
end do
```



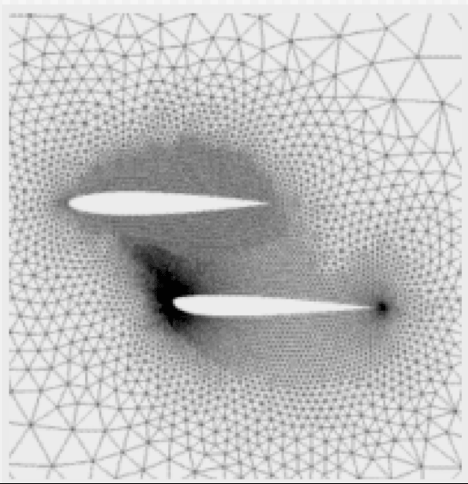
- Align row of A and column of B to avoid communication in the inner k loop



23

Irregular Access Patterns

- Example of an unstructured grid representing the pressure distribution on two airfoils. Image from http://fun3d.larc.nasa.gov/example-23.html.



24

Inspector/Executor

- Impossible to replicate/communicate overlap area for irregular access patterns
- Introduce an Inspector phase that "inspects" the data to be used, identify non-local accesses and produces a communication schedule
- Executor performs computation on local data
- Could of course also be applied to regular overlaps (stencils) but has higher overhead

25

Some things look inherently sequential

- Like …
- Producing the sum of n elements
- Producing the partial sum in an array

26

Reduce and Scan

- Reduce combines a set of values to produce a single value

```
x=0
for i=0,n-1
  x=x+a[i]
end for
```

- Scan (parallel prefix) is an operation that performs a sequential operation in parts and carries along the intermediate results. Often found in loop iterations

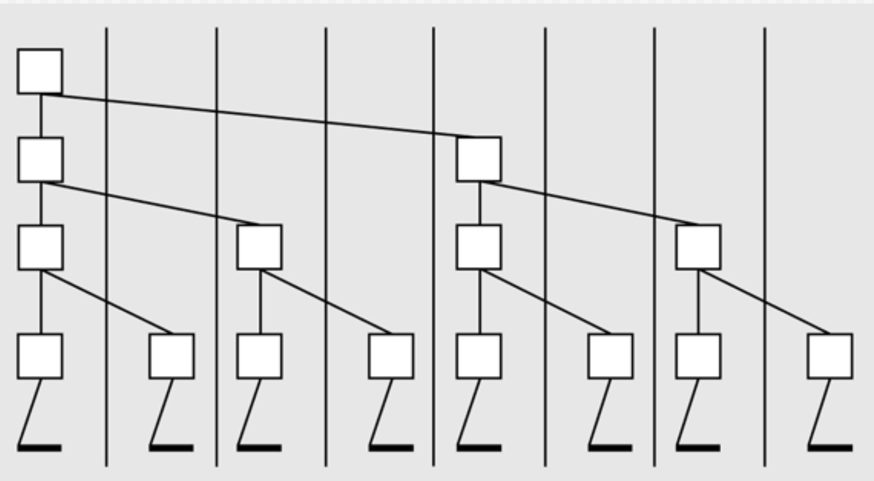
```
y[0]=a[0]
for i=0,n-1
  y[i]=y[i-1]+a[i]
end for
```

- Compute locally on local subsets and use tree to combine results
  - Each process includes a local variable “tally” for storing local results
  - This is possible as the operand is associative

27

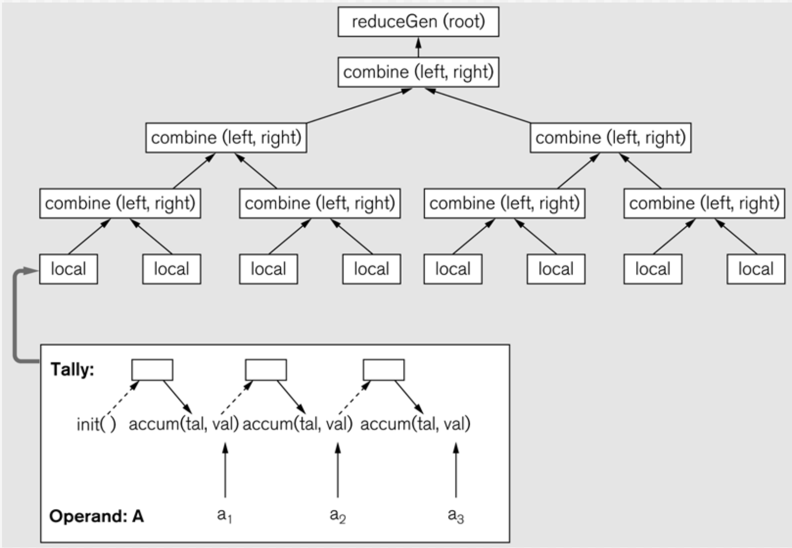
Trees

Computation can be done in a tree like fraction



28

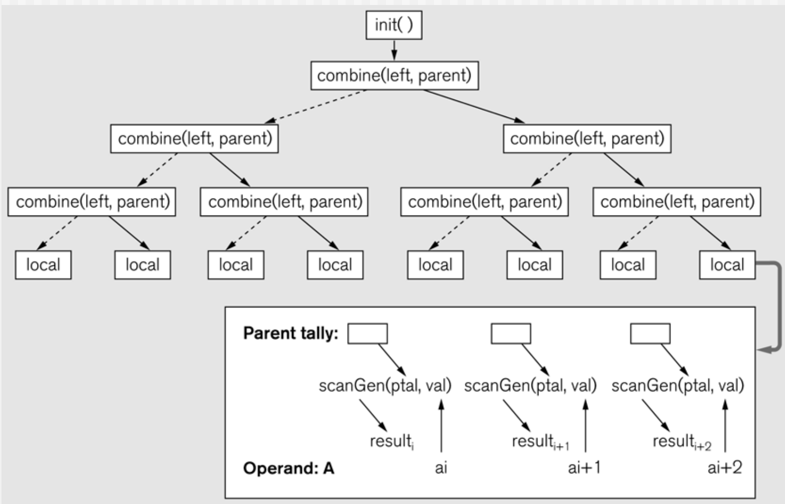
Generalized Reduce



29

Generalized Scan

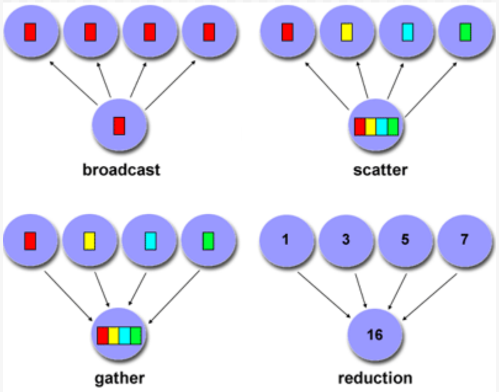
- First reduce, then intermediate results are propagated down



30

Collective Communications

- Same patterns often arise in communication
  - Communication libraries provide optimized implementations



31

Dynamic Work Assignments

- Static domain or functional decompositions don't work well for dynamic problems
  - Workload only known at runtime
- Work queue
  - Data structure for dynamically assigning work to processes
  - Processes can insert (producer) and remove (consumer) work items
  - If there are enough producers work queues deliver good load balance
  - Issues: problems may never be computed if there are too many producers (LIFO)
    - Different priorities, FIFO, aging …
  - Can have multiple queues for scalability

32

Summary

- Basic concepts of functional and spatial decomposition
- Different strategies
- Common patterns

33

Performance Considerations

Communication is quite costly

- Overlapping communication and computation
  - Hide communication latency
  - Increases program complexity
  - *Send Early, Receive Late, Don't Ask but Tell.*
- Redundant computations
  - Perform simple computations on all processors rather than communicate the data
- Data privatization
  - Use additional memory for local variables instead of global variables that require synchronization
  - Useful for anti- and output dependencies

34

Performance Considerations Cont’d

- Parallelize Overhead
  - E.g. Reductions, broadcasts
- Load balance vs. Overhead
  - Fine grained parallelism often allows better load balancing but increases communication/synchronization overhead
- Granularity trade-offs
  - Increase granularity to avoid overhead
  - Batch individual data communications into larger arrays

35

How to formulate Parallelism?

- Fixed Parallelism
  - Formulate the problem such that it can exploit exactly x-way parallelism (x typically the size of the machine available)
- Unlimited Parallelism
  - Try to exploit all possible parallelism available in the problem
  - Can lead to situations where available parallelism is much larger than the available hardware
- What’s the problem with these approaches?

36

Scalable Parallelism

- Determine how components of the problem (data structures, work load, etc.) grow
- Identify a set of *substantial* subproblems to which natural units of work are assigned and solved as *independently* as possible.
- Focus on hot spots - avoid parallel activities with trivial amounts of work.

37

Example: Alphabetizing

- Put a list of records of size N into alphabetic order
- Unlimited Parallelism
  - Maximum number of parallel activities is parallel pair-wise compare
  - Parallelism degree is N/2
  - But might involve a lot of copies (smallest entry at the end of the list needs to propagate through whole list)
- Fixed Parallelism
  - Use 26 processes to sort words beginning with the same letter independently
  - Much better communication behavior
  - Parallelism degree is 26
  - Load imbalance can be an issue

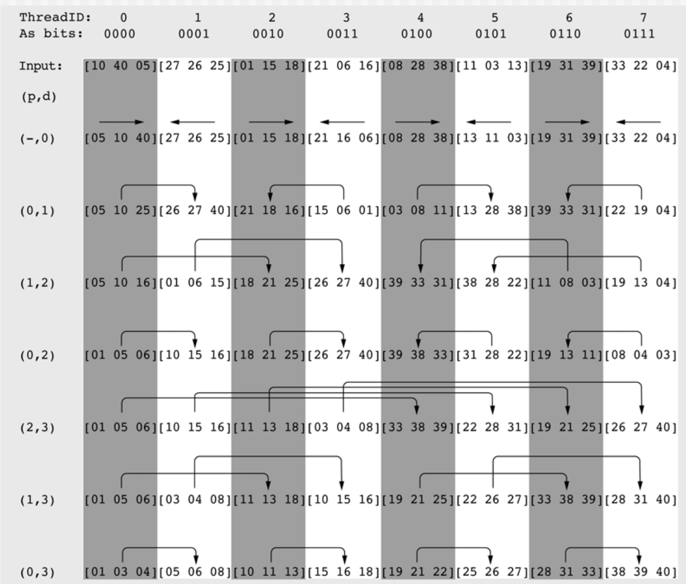
38

Example Continues

- Scalable Parallelism
  - Batcher’s Bitonic Sort (parallel scalable version of mergesort)
  - Each process sorts a number of local records internally either ascending or descending
  - Partial results are being merged in a tree-like fashion
  - Parallelism degree is P provided P is a power of two
  - More complex implementation

39

Batcher’s Bitonic Sort



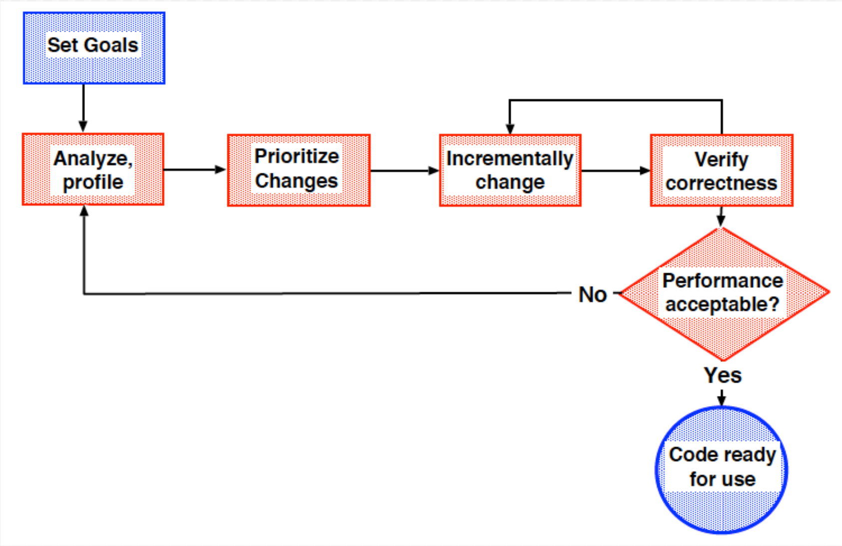
40

How to Start Parallel Programming

- Incremental development
  - Parallel programming is complex
  - Move from one working version to another
- Focus on Data Structure first
  - They will dominate performance
- Identify bottlenecks and hot spots
  - Where is most time spent
  - Use performance tools to identify bottlenecks
  - Compute trivial things sequentially or replicate computation
- Be willing to write extra code
  - To eliminate race conditions, to facilitate testing, instrumentation
- Use existing algorithms and implementations
  - A lot of work went into development of parallel algorithms and highly efficient implementations of common problems - reuse them

41

Parallelization Process



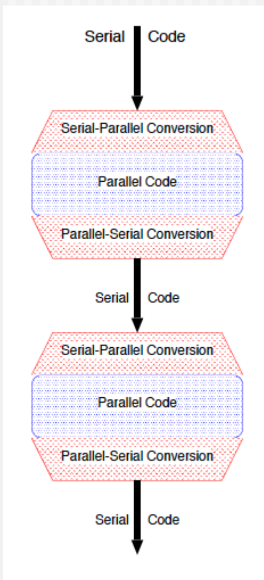
42

Incremental Process

- Shared Memory
  - Incremental parallelization easiest on shared memory
  - Portions not parallelized will be slow but at least correct
    - Major advantage of developing on shared memory over distributed memory
- Distributed Memory
  - More difficult but still possible
  - First steps
    - Coarse grained profiling
    - Map data structures
  - Start with all data global
  - Build sequential - parallel -sequential conversion routines
  - Verify correctness by showing sequential - parallel - sequential = sequential

43

Incremental DM Parallelization



44

If it isn't working well

Check the algorithm

- The original program probably wasn't written with parallelism in mind
- See if there is a more parallelizable approach
- Sometimes parallelizable approaches aren't the most efficient ones available for serial computers but that is OK if you are going to use many processors (sometimes).
- And remember Gene Amdahl:

Efficient massive parallelism is difficult!

45

Summary and Outlook

- Task and Data Parallelism
- Common Strategies
- The impact of data - a closer look

46

Further Readings

- **Principles of Parallel Programming,** Calvin Lin and Lawrence Snyder
  - Part 2 - Parallel Abstractions
- **Introduction to Parallel Computing,** https://computing.llnl.gov/tutorials/parallel_comp/

47

Exercises

16. Does quicksort provide data parallelism or task parallelism? Explain.
17. Does a chess program provide data parallelism or task parallelism? Explain
18. Discuss relative benefits/drawbacks of block and cyclic distributions. Look at granularity (and what that means), communication patterns, load balancing, …
19. Why is data parallelism typically offering a larger degree of parallelism than task parallelism
20. What is an "embarrassingly" parallel problem?

48