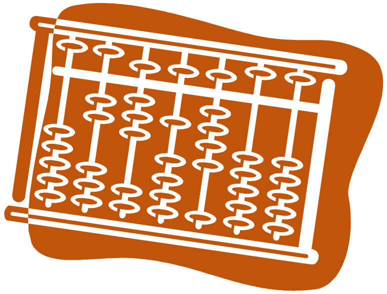
Parallel Programming Principle and Practice

Lecture 5 — Parallel Programming: Performance



Jin, Hai

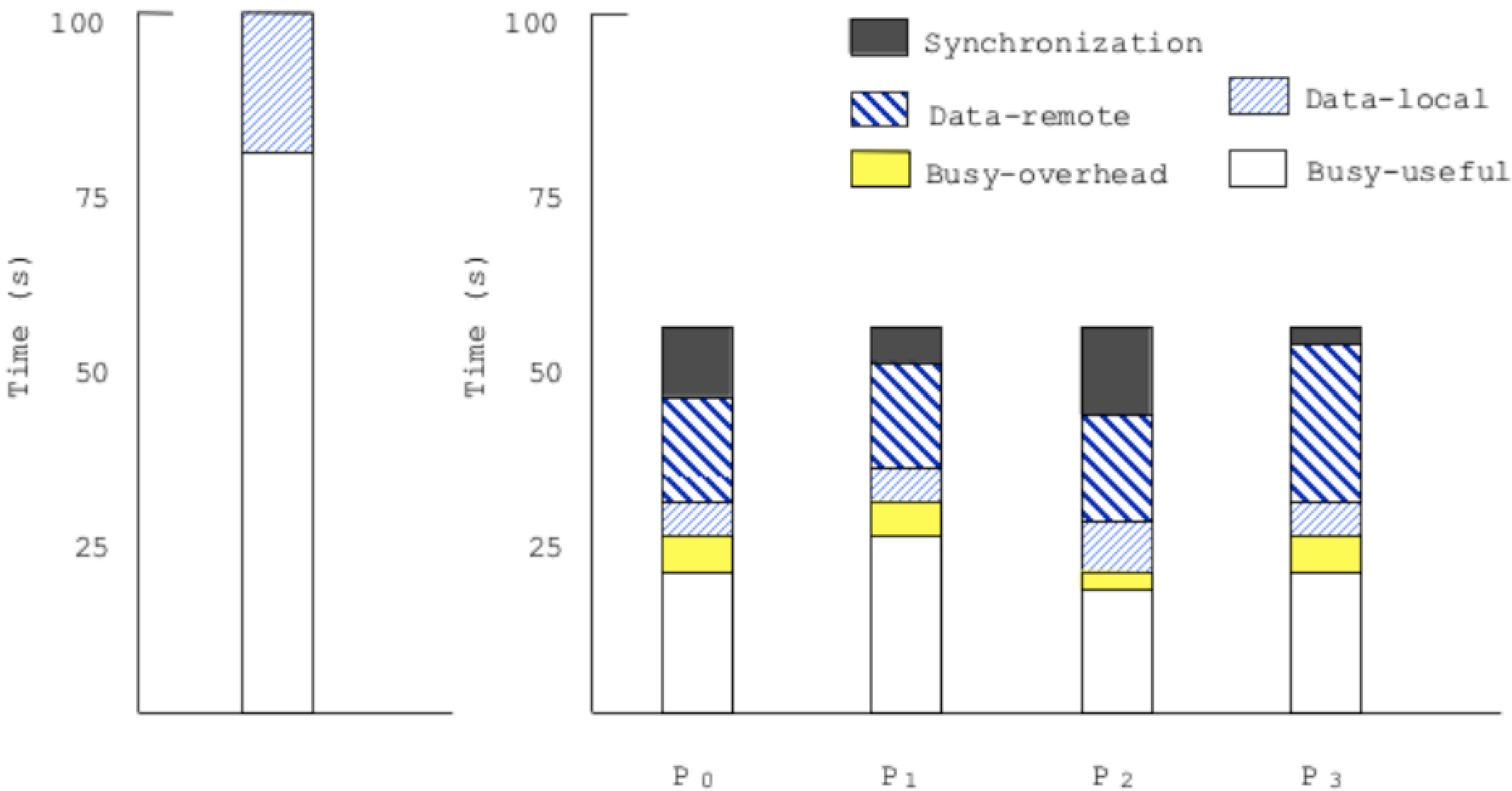
School of Computer Science and Technology

Huazhong University of Science and Technology

Outline

- Components of execution time as seen by processor
- Partitioning for performance
- Relationship of communication, data locality and architecture
- Orchestration for performance

Processor-Centric Perspective



Outline

- Components of execution time as seen by processor
- Partitioning for performance
- Relationship of communication, data locality and architecture
- Orchestration for performance

Partitioning for Performance

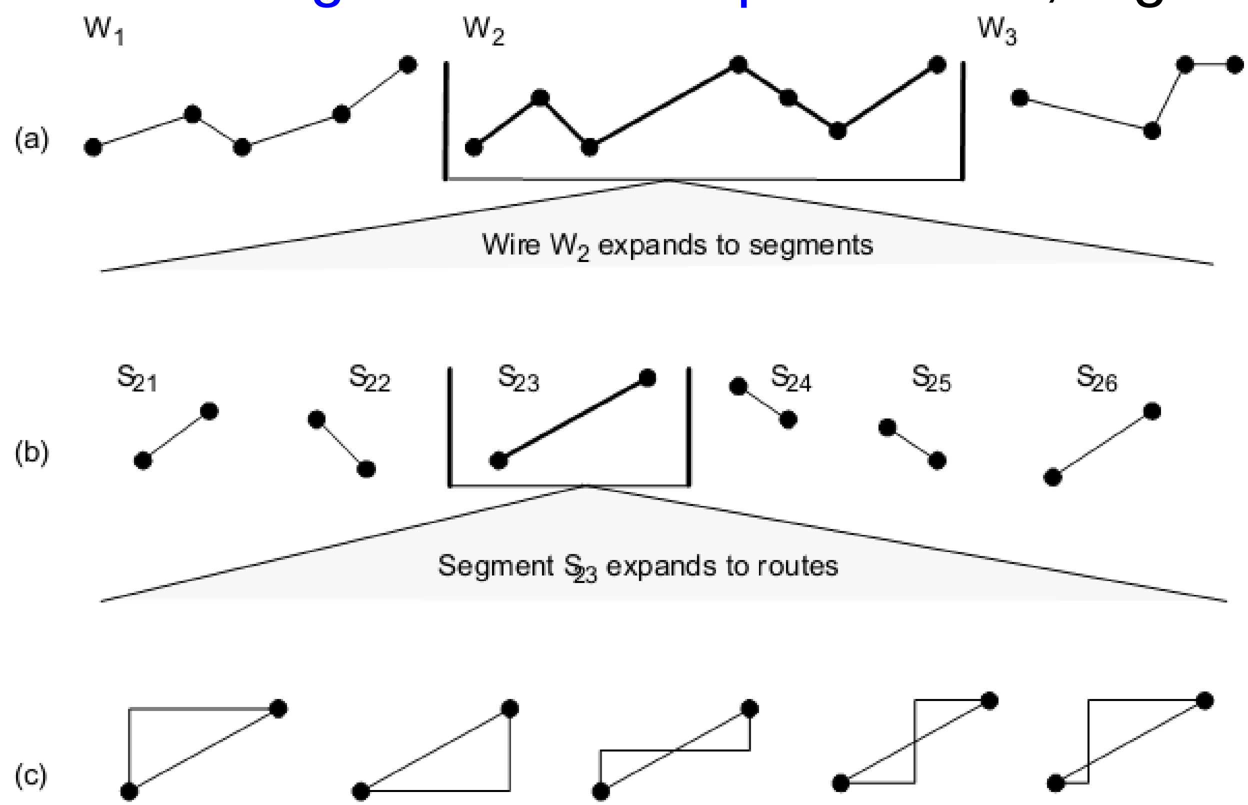
- Balancing the workload and reducing wait time at synch points
- Reducing inherent communication
- Reducing extra work

Load Balance and Synch Wait Time

- Limit on speedup: $Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{\text{Max Work on any Processor}}$
  - Work includes data access and other costs
  - Not just equal work, but must be busy at same time
- Four parts to load balance and reducing synch wait time
  - Identify enough concurrency
  - Decide how to manage it
  - Determine the granularity at which to exploit it
  - Reduce serialization and cost of synchronization

Identifying Concurrency

- Techniques seen for equation solver
  - Loop structure, fundamental dependences, new algorithms
- *Data Parallelism* versus *Function Parallelism*
- Often see orthogonal levels of parallelism; e.g. VLSI routing



华中科技大学

Identifying Concurrency

- Function parallelism
  - entire large tasks (procedures) that can be done in parallel on same or different data
    - e.g. different independent grid computations in Ocean
    - e.g. pipelining, as in video encoding /decoding, or polygon rendering
  - degree usually modest and does not grow with input size
  - difficult to load balance
  - often used to reduce synch between data parallel phases

Identifying Concurrency

- Most scalable programs data parallel
- Data parallelism
  - Similar parallel operation sequences performed on elements of large data structures
    - e.g ocean equation solver, pixel-level image processing
  - Such as resulting from parallelization of loops
  - Usually easy to load balance (e.g ocean equation solver)
  - Degree of concurrency usually increase with input or problem size. e.g $O(n^2)$ in equation solver example

Load Balance and Synch Wait Time

- Limit on speedup: $Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{\text{Max Work on any Processor}}$
  - Work includes data access and other costs
  - Not just equal work, but must be busy at same time
- Four parts to load balance and reducing synch wait time
  - Identify enough concurrency
  - Decide how to manage it
  - Determine the granularity at which to exploit it
  - Reduce serialization and cost of synchronization

Decide How to Manage Concurrency

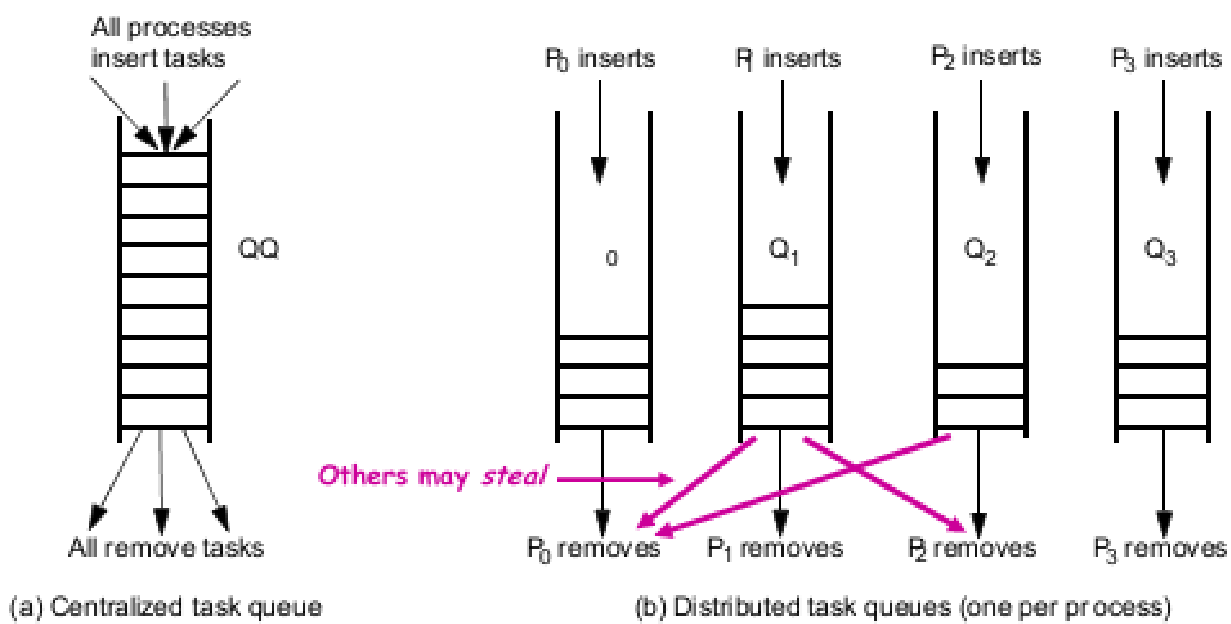
- *Static* versus *Dynamic* techniques
- Static
  - Algorithmic assignment based on input; will not change
  - Low runtime overhead
  - Computation must be predictable
  - Preferable when applicable (except in multiprogrammed or heterogeneous environment)
- Dynamic
  - Adapt at runtime to balance load
  - Can increase communication and reduce locality
  - Can increase task management overheads

Dynamic Assignment

- Profile-based (semi-static)
  - Profile work distribution at runtime, and repartition dynamically
  - Applicable in many computations, e.g. some graphics
- Dynamic Tasking
  - Deal with unpredictability in program or environment (e.g. Raytrace)
    - computation, communication, and memory system interactions
    - multiprogramming and heterogeneity
    - used by runtime systems and OS too
  - Pool of tasks; take and add tasks until done
  - e.g. “self-scheduling” of loop iterations (shared loop counter)

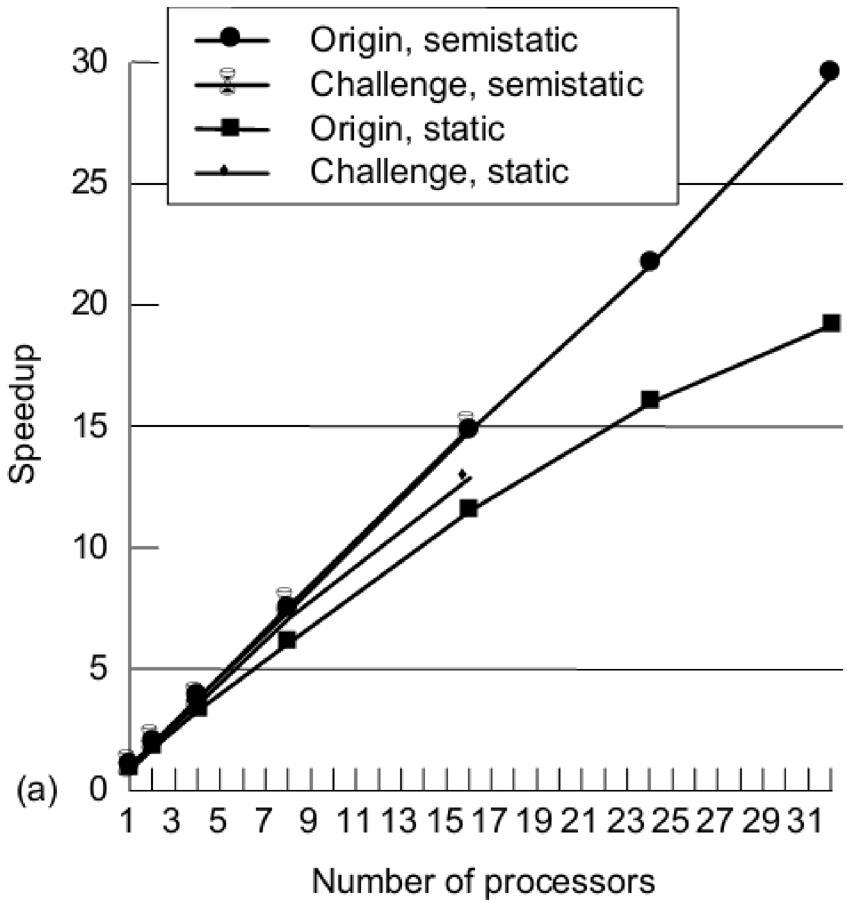
Dynamic Tasking with Task Queues

- Centralized versus distributed queues
- Task stealing with distributed queues
  - Can compromise communication and locality, and increase synchronization
  - Whom to steal from, how many tasks to steal, ...
  - Termination detection
  - Maximum imbalance related to size of task

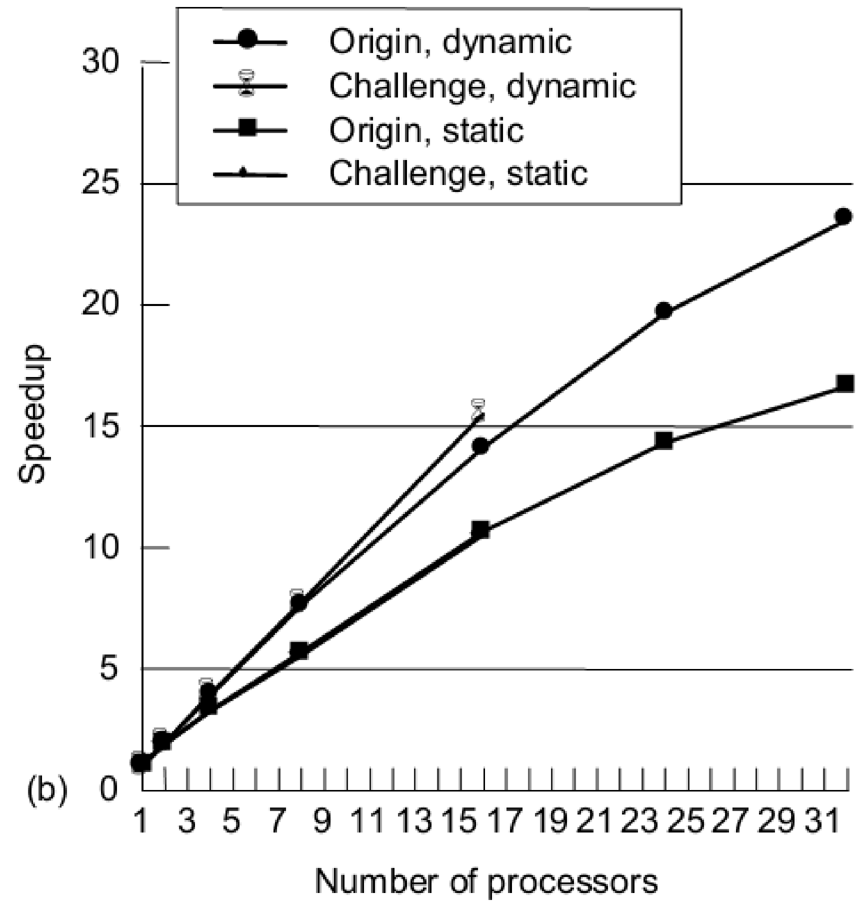


Impact of Dynamic Assignment

- On SGI Origin 2000 (cache-coherent shared memory)



Barnes-Hut



Raytrace

Load Balance and Synch Wait Time

- Limit on speedup: $Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{\textit{Max}\text{ Work on any Processor}}$
  - Work includes data access and other costs
  - Not just equal work, but must be busy at same time

- Four parts to load balance and reducing synch wait time
  - Identify enough concurrency
  - Decide how to manage it
  - Determine the granularity at which to exploit it
  - Reduce serialization and cost of synchronization

Determining Task Granularity

- Task granularity: amount of work associated with a task
- General rule
  - Coarse-grained => often less load balance
  - Fine-grained=> more overhead; often more communication & contention
- Communication & contention actually affected by assignment, not size
  - Overhead by size itself too, particularly with task queues

Load Balance and Synch Wait Time

- Limit on speedup: $Speedup_{problem}(p) \le \frac{\text{Sequential Work}}{\textit{Max}\text{ Work on any Processor}}$
  - Work includes data access and other costs
  - Not just equal work, but must be busy at same time

- Four parts to load balance and reducing synch wait time
  - Identify enough concurrency
  - Decide how to manage it
  - Determine the granularity at which to exploit it
  - Reduce serialization and cost of synchronization

Reducing Serialization

- Careful about assignment and orchestration (including scheduling)
- Event synchronization
  - Reduce use of conservative synchronization
    - e.g. point-to-point instead of barriers, or granularity of pt-to-pt
  - But fine-grained synch more difficult to program, more synch ops.
- Mutual exclusion
  - Separate locks for separate data
    - e.g. locking records in a database: lock per process, record, or field
    - lock per task in task queue, not per queue
    - finer grain => less contention/serialization, more space, less reuse
  - Smaller, less frequent critical sections
    - Do not do reading/testing in critical section, only modification
    - e.g. searching for task to dequeue in task queue, building tree
  - Stagger critical sections in time

Partitioning for Performance

- Balancing the workload and reducing wait time at synch points
- Reducing inherent communication
- Reducing extra work

Reducing Inherent Communication

- Communication is expensive!
- Measure: communication to computation ratio
- Focus here on inherent communication
  - Determined by assignment of tasks to processes
  - Later see that actual communication can be greater
- Assign tasks that access same data to same process
- Solving communication and load balance NP-hard in general case
- But simple heuristic solutions work well in practice
  - Applications have structure

Implications of Communication-to-Computation Ratio

- If denominator is execution time, ratio gives average bandwidth needs
- If denominator is operation count, gives extremes in impact of latency and bandwidth
  - Latency: assume no latency hiding
  - Bandwidth: assume all latency hidden
  - Reality is somewhere in between
- Actual impact of communication depends on structure & cost as well

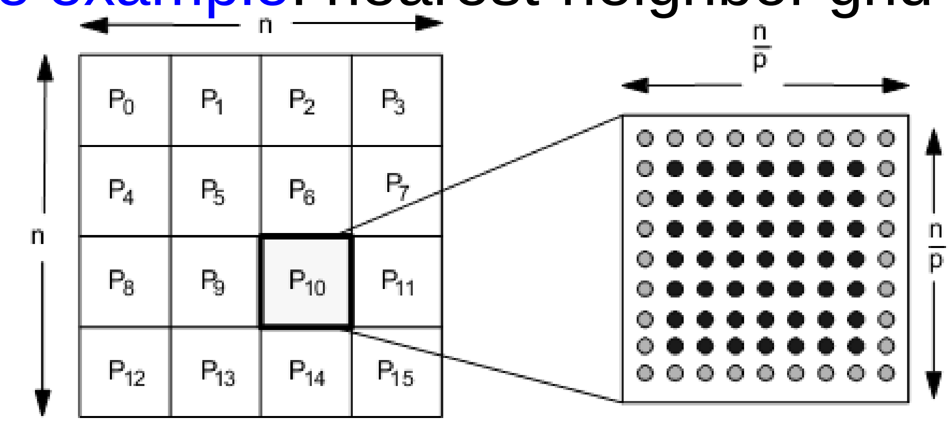
$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\textit{Max}\,(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost})}$$

  - Need to keep communication balanced across processors as well

Domain Decomposition

- Works well for scientific, engineering, graphics, …applications
- Exploits local-biased nature of physical problems
  - Information requirements often short-range

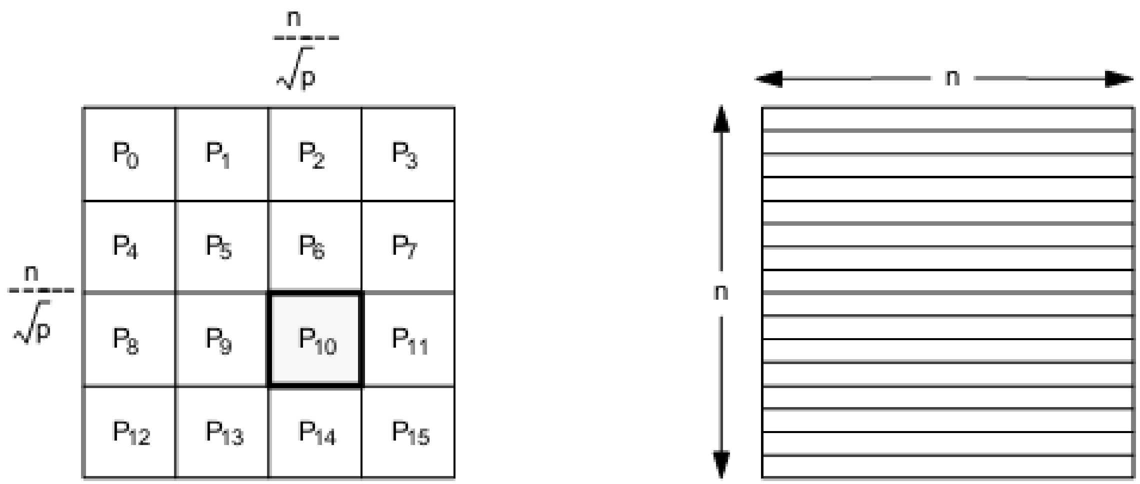
    Or long-range but fall off with distance
- Simple example: nearest-neighbor grid computation



  - Depends on *n,p* : decreases with *n*, increases with *p*

Domain Decomposition

- Best domain decomposition depends on information requirements
- Nearest neighbor example: block versus strip decomposition



- Comm to comp: $\frac{4*\sqrt{p}}{n}$ for block, $\frac{2*p}{n}$ for strip
  - Retain block from here on
- Application dependent: strip may be better in other cases
  - E.g. particle flow in tunnel

Finding a Domain Decomposition

- Static, by inspection
  - Must be predictable: grid example above, and Ocean
- Static, but not by inspection
  - Input-dependent, require analyzing input structure
  - e.g. sparse matrix computations, data mining
- Semi-static (periodic repartitioning)
  - Characteristics change but slowly; e.g. Barnes-Hut
- Static or semi-static, with dynamic task stealing
  - Initial decomposition, but highly unpredictable; e.g. ray tracing

Relation to Load Balance

- Scatter Decomposition, e.g. initial partition in Raytrace

12	
3	4

Domain decomposition

12		12		12		12	
3	4	3	4	3	4	3	4
12		12		12		12	
3	4	3	4	3	4	3	4
12		12		12		12	
3	4	3	4	3	4	3	4
12		12		12		12	
3	4	3	4	3	4	3	4

Scatter decomposition

Preserve locality in task stealing

- Steal large tasks for locality, steal from same queues, ...

Partitioning for Performance

- Balancing the workload and reducing wait time at synch points
- Reducing inherent communication
- Reducing extra work

Reducing Extra Work

- Common sources of extra work
  - Computing a good partition
    - e.g. partitioning in Barnes-Hut or sparse matrix
  - Using redundant computation to avoid communication
  - Task, data and process management overhead
    - applications, languages, runtime systems, OS
  - Imposing structure on communication
    - coalescing messages, allowing effective naming
- Architectural Implications
  - Reduce need by making communication and orchestration efficient

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max}\,(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}$$

Outline

- Components of execution time as seen by processor
- Partitioning for performance
- Relationship of communication, data locality and architecture
- Orchestration for performance

Limitations of Algorithm Analysis

- Inherent communication in parallel algorithm is not all
  - artifactual communication caused by program implementation and architectural interactions can even dominate
  - thus, amount of communication not dealt with adequately
- Cost of communication determined not only by amount
  - also how communication is structured
  - and cost of communication in system
- Both architecture-dependent, and addressed in orchestration step
- To understand techniques, first look at system interactions

What is a Multiprocessor?

- A collection of communicating processors
  - View taken so far
  - Goals: balance load, reduce inherent communication and extra work
- A multi-cache, multi-memory system
  - Role of these components essential regardless of programming model
  - Programming model and communication abstraction affect specific performance tradeoffs
- Most of remaining performance issues focus on second aspect

Memory-Oriented View

- Multiprocessor as extended memory hierarchy
  - as seen by a given processor
- Levels in extended hierarchy
  - Registers, caches, local memory, remote memory (topology)
  - Glued together by communication architecture
  - Levels communicate at a certain granularity of data transfer
- Need to exploit spatial and temporal locality in hierarchy
  - Otherwise extra communication may also be caused
  - Especially important since communication is expensive

Extended Hierarchy

- Idealized view: local cache hierarchy + single main memory
- But reality is more complex
  - Centralized Memory: caches of other processors
  - Distributed Memory: some local, some remote; + network topology
  - Management of levels
    - caches managed by hardware
    - main memory depends on programming model
      - SAS: data movement between local and remote transparent
      - message passing: explicit
  - Levels closer to processor are lower latency and higher bandwidth
  - Improve performance through architecture or program locality
  - Tradeoff with parallelism; need good node performance and parallelism

Artifactual Communication in Extended Hierarchy

- Accesses not satisfied in local portion cause communication
  - Inherent communication, implicit or explicit, causes transfers
    - determined by program
  - Artifactual communication
    - determined by program implementation and architecture interactions
    - poor allocation of data across distributed memories
    - unnecessary data in a transfer
    - unnecessary transfers due to system granularities
    - redundant communication of data
    - finite replication capacity (in cache or main memory)
  - Inherent communication assumes unlimited capacity, small transfers, perfect knowledge of what is needed

Outline

- Components of execution time as seen by processor
- Partitioning for performance
- Relationship of communication, data locality and architecture
- Orchestration for performance

Orchestration for Performance

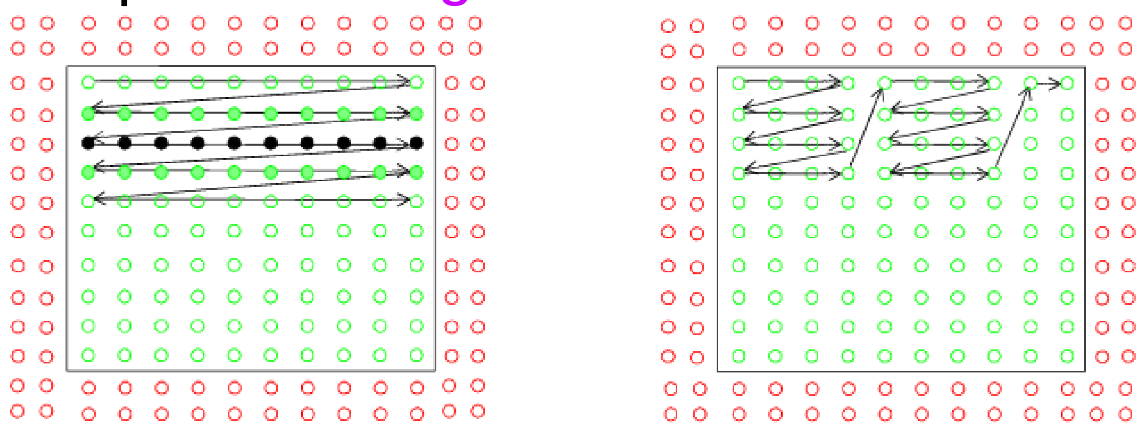
- Reducing amount of communication
  - Artifactual: exploit spatial, temporal locality in extended hierarchy
  - Inherent: change logical data sharing patterns in algorithm
- Structuring communication to reduce cost
- Let's examine techniques for both

Reducing Artifactual Communication

- Message passing model
  - Communication and replication are both explicit
  - Even artifactual communication is in explicit messages
- Shared address space model
  - More interesting from an architectural perspective
  - Occurs transparently due to interactions of program and system
    - sizes and granularities in extended memory hierarchy
- Use shared address space to illustrate issues

Exploiting Temporal Locality

- Structure algorithm so that working sets map well to hierarchy
  - often techniques to reduce inherent communication do well here
  - schedule tasks for data reuse once assigned
- Multiple data structures in same phase
  - e.g. database records: local versus remote
- Solver example: blocking



(a) Unblocked access pattern in a sweep (b) Blocked access pattern with B = 4

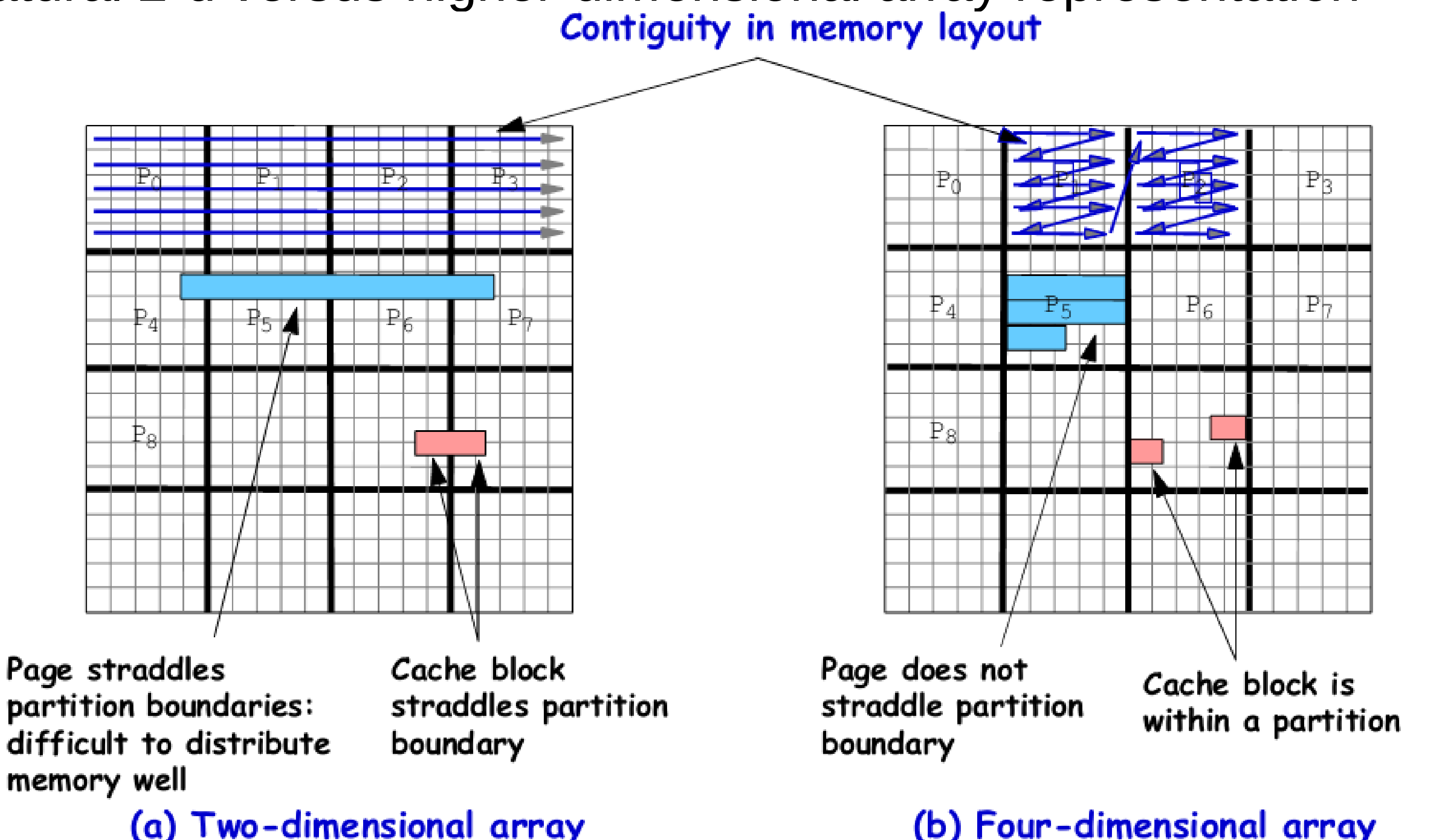
- More useful when $O(n^{k+1})$ computation on $O(n^k)$ data
  - many linear algebra computations (factorization, matrix multiply)

Exploiting Spatial Locality

- Besides capacity, granularities are important
  - Granularity of allocation
  - Granularity of communication or data transfer
  - Granularity of coherence
- Major spatial-related causes of artifactual communication
  - Conflict misses
  - Data distribution/layout (allocation granularity)
  - Fragmentation (communication granularity)
  - False sharing of data (coherence granularity)
- All depend on how spatial access patterns interact with data structures
  - Fix problems by modifying data structures, or layout/alignment
- Examine later in context of architectures
  - one simple example here: data distribution in SAS solver

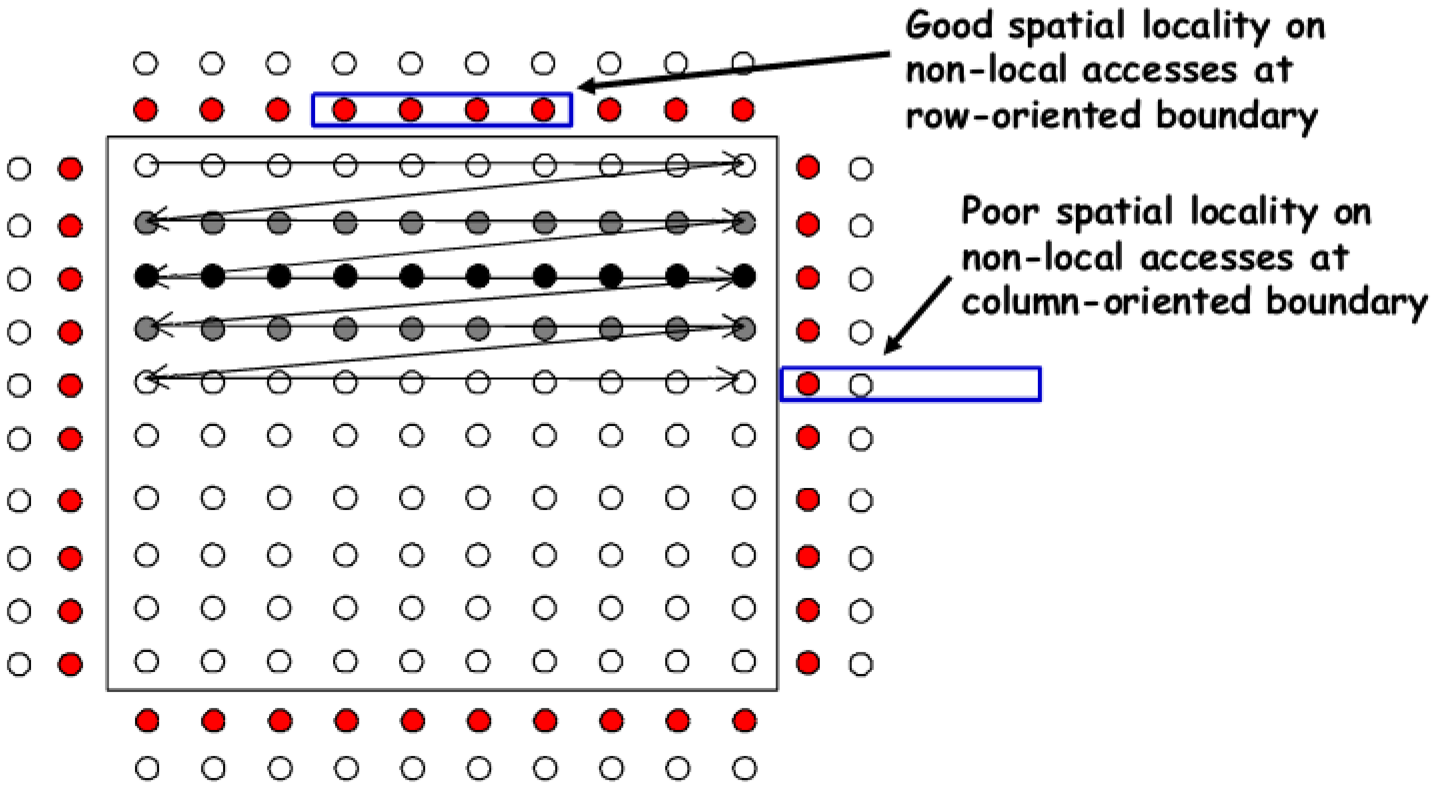
Spatial Locality Example

- Repeated sweeps over 2-d grid, each time adding 1 to elements
- Natural 2-d versus higher-dimensional array representation



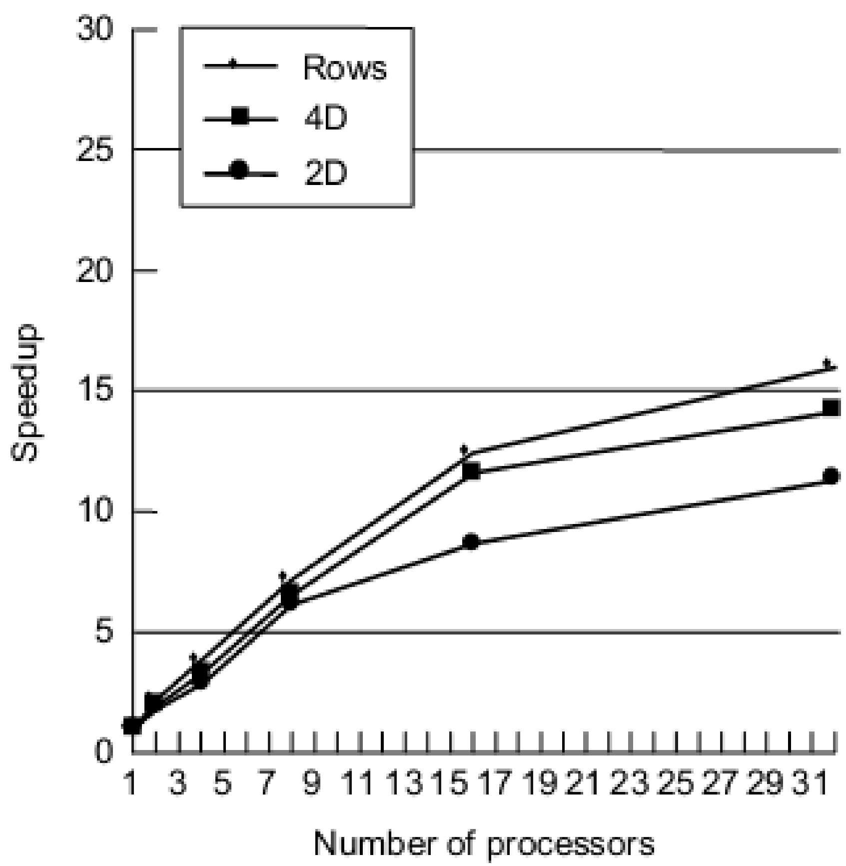
Tradeoffs with Inherent Communication

- Partitioning grid solver: blocks versus rows
  - Blocks still have a spatial locality problem on remote data
  - Rows can perform better despite worse inherent c-to-c ratio

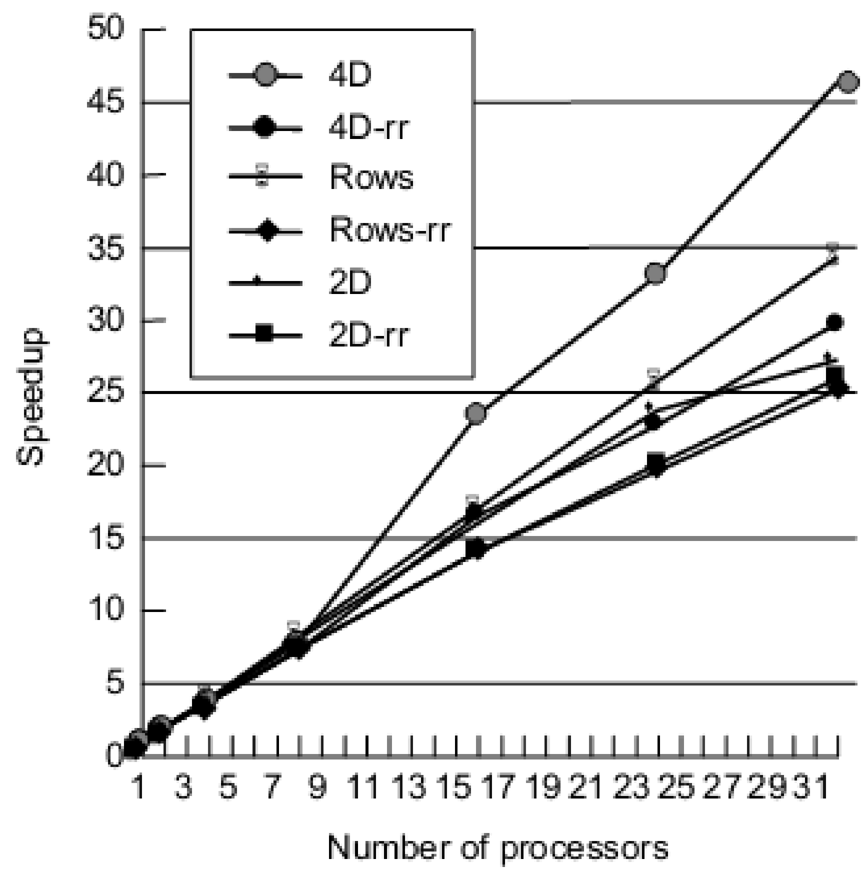


Example Performance Impact

- Performance measured on an SGI Origin2000



Ocean, 514x514 grids



Solver Kernel, 12K x 12K grid

Structuring Communication

- Given amount of communication, goal is to reduce cost
- Cost of communication as seen by process

$$C = f * \left(o + l + \frac{n_c/m}{B} + t_c - overlap\right)$$

  - f = frequency of messages
  - o = overhead per message (at both ends)
  - l = network delay per message
  - n_c= total data sent
  - m = number of messages
  - B = bandwidth along path (determined by network, NI, assist)
  - t_c= cost induced by content *i* on per message
  - *overlap* = amount of latency hidden by overlap with comp. or comm.
- Portion in parentheses is cost of a message (as seen by processor)
- That portion, ignoring overlap, is latency of a message
- Goal: reduce terms in latency and increase overlap

Reducing Overhead

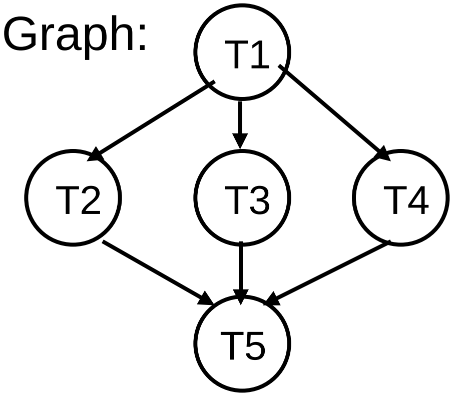
- Can reduce # of messages *m* or overhead per message *o*
- *o* is usually determined by hardware or system software
  - Program should try to reduce *m* by coalescing messages
  - More control when communication is explicit
- Coalescing data into larger messages
  - Easy for regular, coarse-grained communication
  - Can be difficult for irregular, naturally fine-grained communication
    - may require changes to algorithm and extra work
      - coalescing data and determining what and to whom to send

Reducing Network Delay

- Network delay component= $f * h * t_h$
  - h = number of hops traversed in network
  - t_h= link + switch latency per hop
- Reducing f : communicate less, or make messages larger
- Reducing h
  - Map communication patterns to network topology
    - e.g. nearest-neighbor on mesh and ring; all-to-all
  - How important is this?
    - used to be major focus of parallel algorithms
    - depends on number of processors, how t_h, compares with other components
    - less important on modern machines
      - ✓ overheads, processor count, multiprogramming

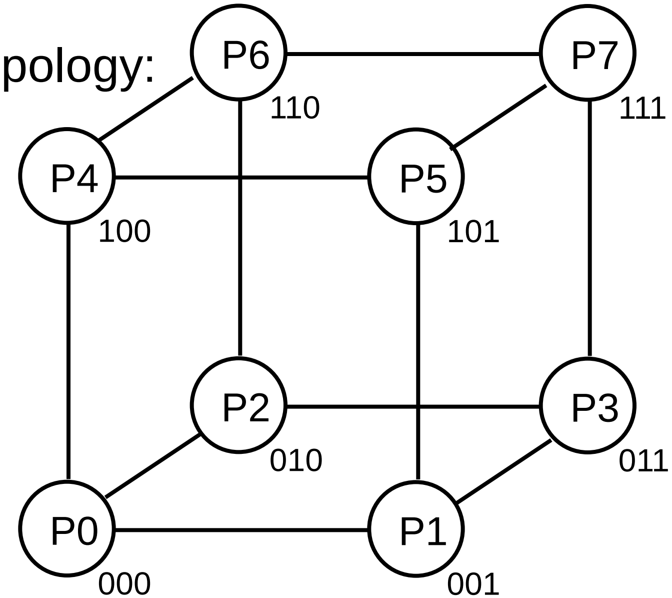
Mapping of Task Communication Patterns to Topology

Task Graph:



Parallel System Topology:

3D Binary Hypercube



Poor Mapping:

T1 runs on P0
T2 runs on P5
T3 runs on P6
T4 runs on P7
T5 runs on P0

- Communication from T1 to T2 requires 2 hops
  Route: P0-P1-P5
- Communication from T1 to T3 requires 2 hops
  Route: P0-P2-P6
- Communication from T1 to T4 requires 3 hops
  Route: P0-P1-P3-P7
- Communication from T2, T3, T4 to T5
  - similar routes to above reversed (2-3 hops)

Better Mapping:

T1 runs on P0
T2 runs on P1
T3 runs on P2
T4 runs on P4
T5 runs on P0

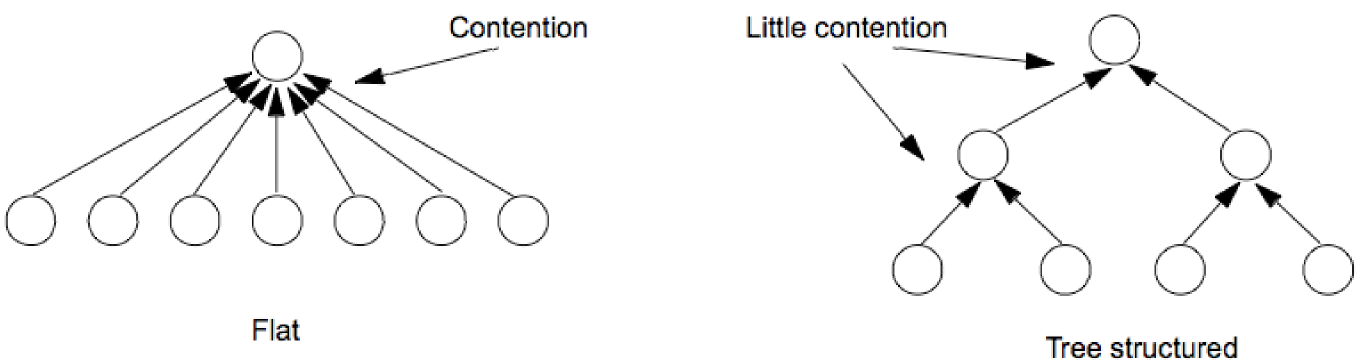
- Communication between any two communicating (dependant) tasks requires just 1 hop

Reducing Contention

- All resources have nonzero occupancy
  - Memory, communication controller, network link, etc.
  - Can only handle so many transactions per unit time
- Effects of contention
  - Increased end-to-end cost for messages
  - Reduced available bandwidth for individual messages
  - Causes imbalances across processors
- Particularly insidious performance problem
  - Easy to ignore when programming
  - Slow down messages that don't even need that resource
    - by causing other dependent resources to also congest
  - Effect can be devastating: *Don't flood a resource!*

Types of Contention

- Network contention and end-point contention *(hot-spots)*
- *Location* and *Module* hot-spots
- *Location*: e.g. accumulating into global variable barrier
  - solution: tree-structured communication



  - In general, reduce burstiness; may conflict with making messages
- *Module*: all-to-all personalized comm. in matrix transpose
  - solution: stagger access by different processors to same node temporally

Overlapping Communication

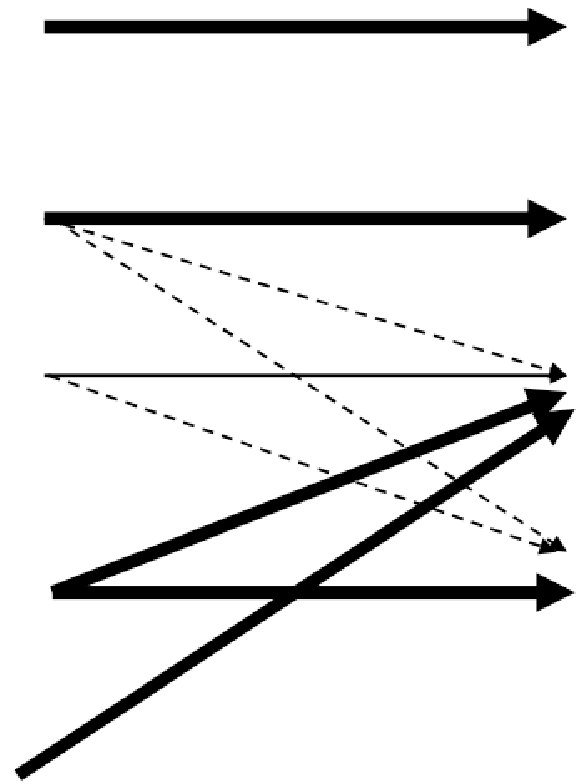
- Cannot afford to stall for high latencies
  - even on uniprocessors!
- Overlap with computation or communication to hide latency
- Requires extra concurrency (*slackness*), higher bandwidth
- Techniques
  - Prefetching
  - Block data transfer
  - Proceeding past communication
  - Multithreading

Summary of Tradeoffs

- Different goals often have conflicting demands
  - Load Balance
    - fine-grain tasks
    - random or dynamic assignment
  - Communication
    - usually coarse grain tasks
    - decompose to obtain locality: not random/dynamic
  - Extra Work
    - coarse grain tasks
    - simple assignment
  - Communication Cost
    - big transfers: amortize overhead and latency
    - small transfers: reduce contention

Relationship between Perspectives

Parallelization step(s)	Performance issue
Decomposition/ assignment/ orchestration	Load imbalance and synchronization
Decomposition/ assignment	Extra work
Decomposition/ assignment	Inherent communication volume
Orchestration	Artifactual communication and data locality
Orchestration/ mapping	Communication structure



Processor time component

- Synch wait
- Busy-overhead
- Data-remote
- Data-local

Summary

$$Speedup_{prob}(p) = \frac{Busy(1) + Data(1)}{Busy_{useful}(p) + Data_{local}(p) + Synch(p) + Data_{remote}(p) + Busy_{overhead}(p)}$$

- Goal is to reduce denominator components
- Both programmer and system have role to play
- Architecture cannot do much about load imbalance or too much communication
- But it can
  - reduce incentive for creating ill-behaved programs (efficient naming, communication and synchronization)
  - reduce artifactual communication
  - provide efficient naming for flexible assignment
  - allow effective overlapping of communication

References

- The content expressed in this chapter comes from
  - Carnegie Mellon University's public course, Parallel Computer Architecture and Programming, (CS 418) (http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/)