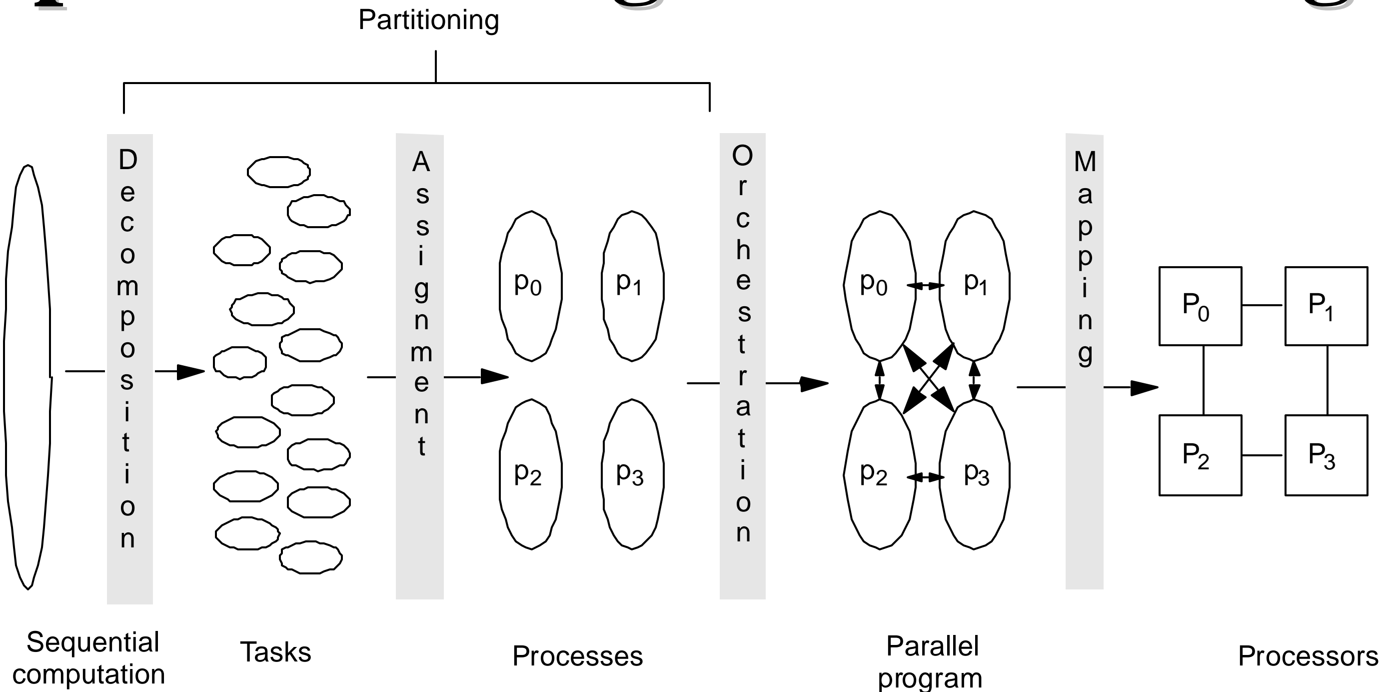
Steps in Creating a Parallel Program



- **4 steps: Decomposition, Assignment, Orchestration, Mapping**
- **Performance Goal of the steps: Minimize resulting execution time by:**
  - **Balancing computations on processors (every processor does the same amount of work).**
  - **Minimizing communication cost and other overheads associated with each step.**

Parallel Programming for Performance

A process of Successive Refinement of the steps

- **Partitioning for Performance:**
  - Load Balancing and Synch Wait Time Reduction
  - Identifying & Managing Concurrency
    - **Static Vs. Dynamic Assignment**
    - **Determining Optimal Task Granularity**
    - **Reducing Serialization**
  - Reducing Inherent Communication
    - **Minimizing** ***communication to computation ratio***
  - Efficient Domain Decomposition
  - Reducing Additional Overheads
- **Orchestration/Mapping for Performance:**
  - Extended Memory-Hierarchy View of Multiprocessors
    - Exploiting Spatial Locality/Reduce Artifactual Communication
    - Structuring Communication
    - Reducing Contention
    - Overlapping Communication

Successive Refinement of Parallel Program Performance

Partitioning is often independent of architecture, and may be done first:

- View machine as a collection of communicating processors
  - Balancing the workload across processes/processors.
  - Reducing the amount of inherent communication.
  - Reducing extra work to find a good assignment.
- Above three issues are conflicting.

Then deal with interactions with architecture (Orchestration, Mapping) :

- View machine as an extended memory hierarchy:
  - Extra communication due to architectural interactions.
  - Cost of communication depends on how it is structured
- This may inspire changes in partitioning.

Partitioning for Performance

- Balancing the workload across processes:
  - reducing wait time at synch points.
- Reducing interprocess inherent communication.
- Reducing extra work needed to find a good assignment.

These algorithmic issues have extreme trade-offs:

- Minimize communication => run on 1 processor.
  => extreme load imbalance.
- Maximize load balance => random assignment of tiny tasks.
  => no control over communication.
- Good partition may imply extra work to compute or manage it

- The goal is to compromise between the above extremes

Load Balancing and Synch Wait Time Reduction

Limit on speedup:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{Max \text{ Work on any Processor}}$$

- **Work includes data access and other costs.**
- **Not just equal work, but must be busy at same time to minimize synch wait time.**

Four parts to load balancing and reducing synch wait time:

1. **Identify enough concurrency.**
2. **Decide how to manage it.**
3. **Determine the granularity at which to exploit it.**
4. **Reduce serialization and cost of synchronization.**

Identifying Concurrency: Decomposition

- Concurrency may be found by:
  - Examining loop structure of sequential algorithm.
  - Fundamental data dependences.
  - Exploit the understanding of the problem to devise algorithms with more concurrency.
- Data Parallelism versus Function Parallelism:
- Data Parallelism:
  - Parallel operation sequences performed on elements of large data structures
  - Such as resulting from parallization of loops.
  - Usually easy to load balance.
  - Degree of concurrency usually increase with input or problem size.

Identifying Concurrency (continued)

Function or Task parallelism:

- **Entire large tasks (procedures) that can be done in parallel on the same or different data.**

  e.g. different independent grid computations in Ocean.

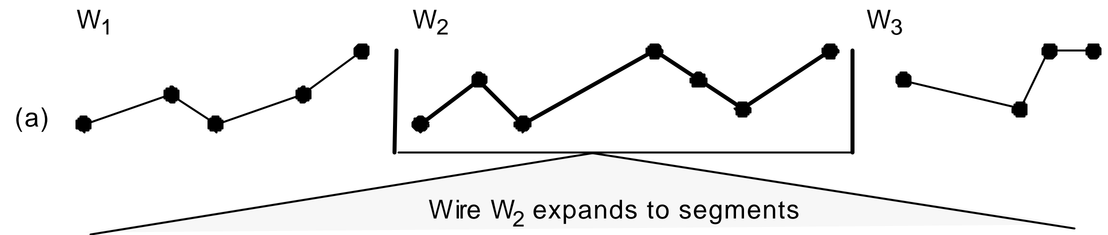
- **Software Pipelining: Different functions or software stages of the pipeline performed on different data:**
  - **As in video encoding/decoding, or polygon rendering.**
- **Degree usually modest and does not grow with input size**
- **Difficult to load balance.**
- **Often used to reduce synch between data parallel phases.**

Most scalable parallel programs: Data parallel (per this loose definition)

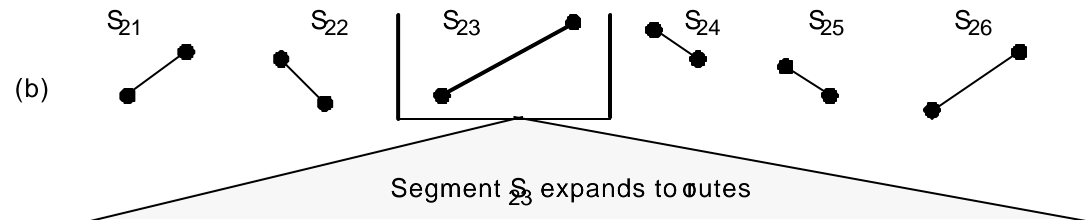
- **Function parallelism reduces synch between data parallel phases.**

Levels of Parallelism in VLSI Routing

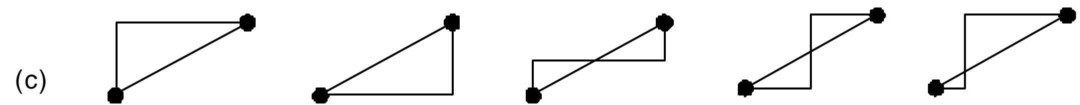
Wire Parallelism



Segment Parallelism



Route Parallelism



Managing Concurrency: Assignment

Goal: Obtain an assignment with a good load balance among tasks (and processors in mapping step)

Static versus *Dynamic* Assignment:

Static Assignment:

- **Algorithmic assignment usually based on input data ; does not change at run time.**
- **Low run time overhead.**
- **Computation must be predictable.**
- **Preferable when applicable.**

Dynamic Assignment:

- **Adapt partitioning at run time to balance load on processors.**
- **Can increase communication cost and reduce data locality.**
- **Can increase run time task management overheads.**

Dynamic Assignment/Mapping

Profile-based (semi-static):

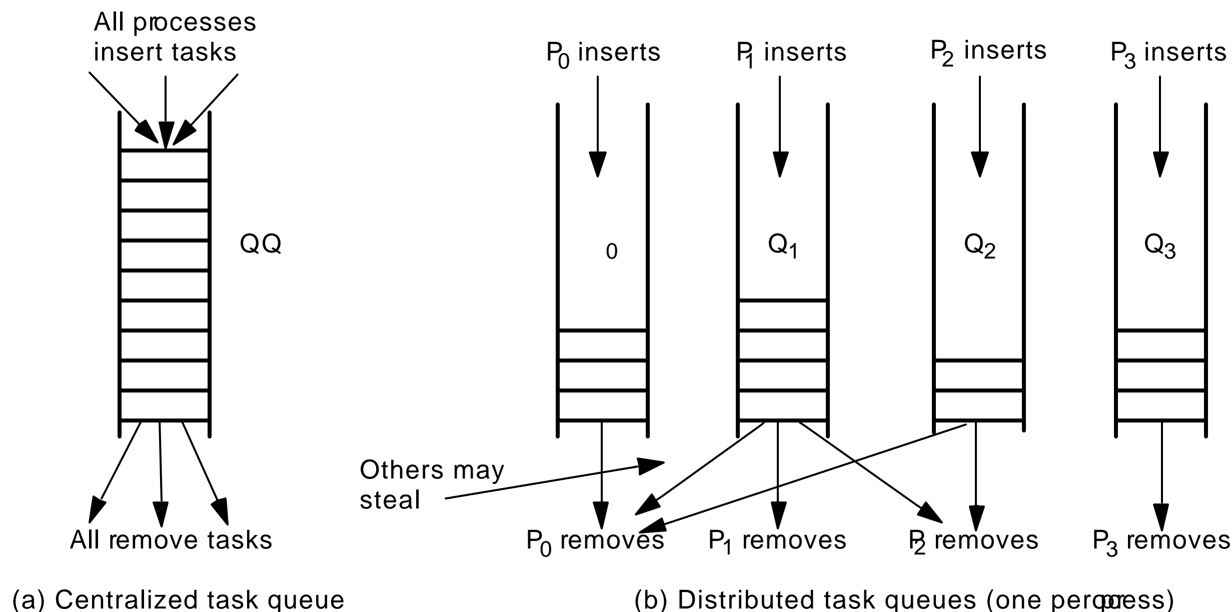
- Profile (algorithm) work distribution initially at runtime, and repartition dynamically.
- Applicable in many computations, e.g. Barnes-Hut, some graphics.

Dynamic Tasking:

- Deal with unpredictability in program or environment (e.g. Ray trace)
  - 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 data locality, and increase synchronization.**
  - **Whom to steal from, how many tasks to steal, ...**
  - **Termination detection (all queues empty).**
  - **Maximum load imbalance possible related to size of task.**

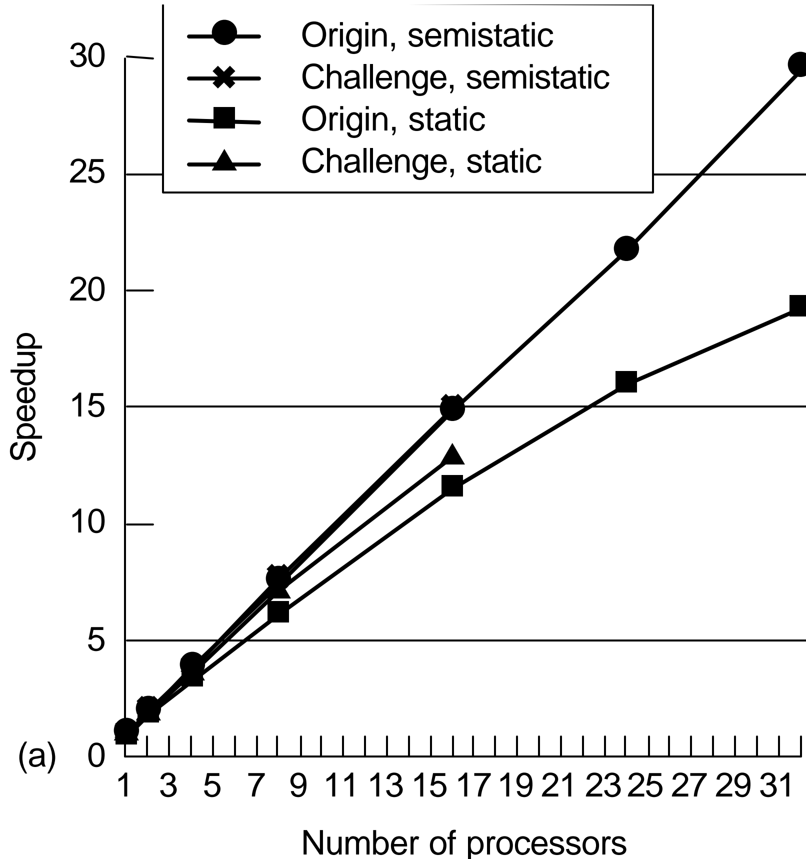


(a) Centralized task queue

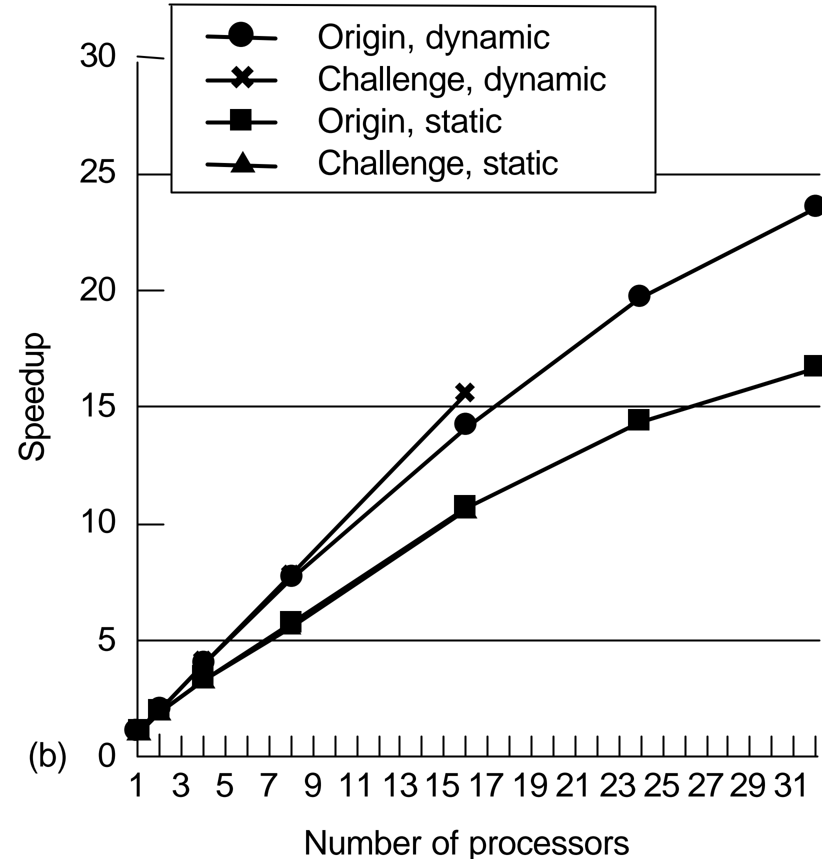
(b) Distributed task queues (one per process)

Impact of Dynamic Assignment

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



Barnes-Hut 512k particle



Ray trace

Assignment: Determining Task Granularity

Recall that parallel task granularity:

Amount of work associated with a task.

General rule:

- **Coarse-grained => often less load balance
less communication**
- **Fine-grained => more overhead; often more
communication , contention**

Communication, contention actually more affected by mapping to processors, not just task size only.

- **Overhead affected by task size itself too, particularly with task queues**

Reducing Serialization

Careful 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 operations.

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
  - No reading/testing in critical section, only modification
  - e.g. searching for task to dequeue in task queue, building tree
- Stagger critical sections in time.

Implications of Load Balancing

Extends speedup limit expression to:

$$Speedup_{problem}(p) \leq \frac{\text{Sequential Work}}{Max\ (\text{Work + Synch Wait Time})}$$

Generally load balancing is the responsibility of software

Architecture can support task stealing and synch efficiently:

- **F*ine-grained* communication, *low-overhead access* to queues**
  - **Efficient support allows smaller tasks, better load balancing**
- **N*aming* logically shared data in the presence of task stealing**
  - **Need to access data of stolen tasks, esp. multiply-stolen tasks**

=> Hardware shared address space advantageous

- **Efficient support for point-to-point communication.**

Reducing Inherent Communication

Measure: *communication to computation ratio (c-to-c ratio)*

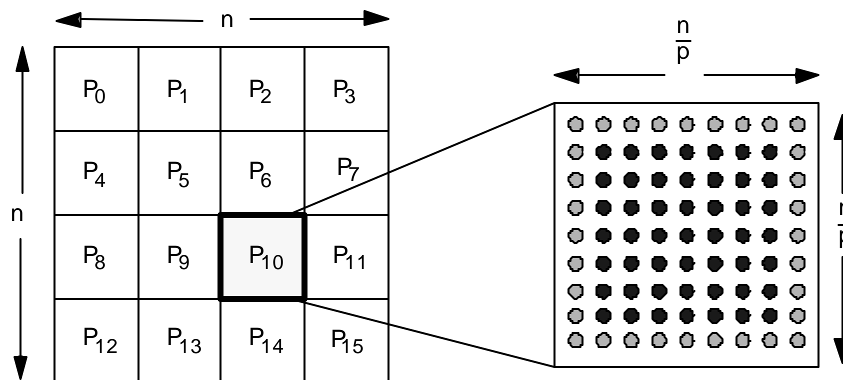
Focus here is on interprocess communication inherent in the problem:

- Determined by assignment of tasks to processes.
- Minimize c-to-c ratio while maintaing a good load balance among processes.
- Actual communication can be greater.

- As much as possible, assign tasks that access same data to same process.
- Optimal solution to reduce communication and achieve an optimal load balance is NP-hard in the general case.
- Simple heuristic solutions may work well in practice:
  - Due to specific dependency structure of applications.

Domain Decomposition

- **Initially used in data parallel scientific computations such as (Ocean) to obtain a good load balance and c-to-c ratio.**
- **The task assignment is achieved by decomposing the physical domain or data set of the problem.**
- **Exploits the 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**



$$Communication = \frac{4n}{\sqrt{p}} \qquad Computation = \frac{n^2}{p}$$

$$C-to-C = \frac{4 \times \sqrt{p}}{n}$$

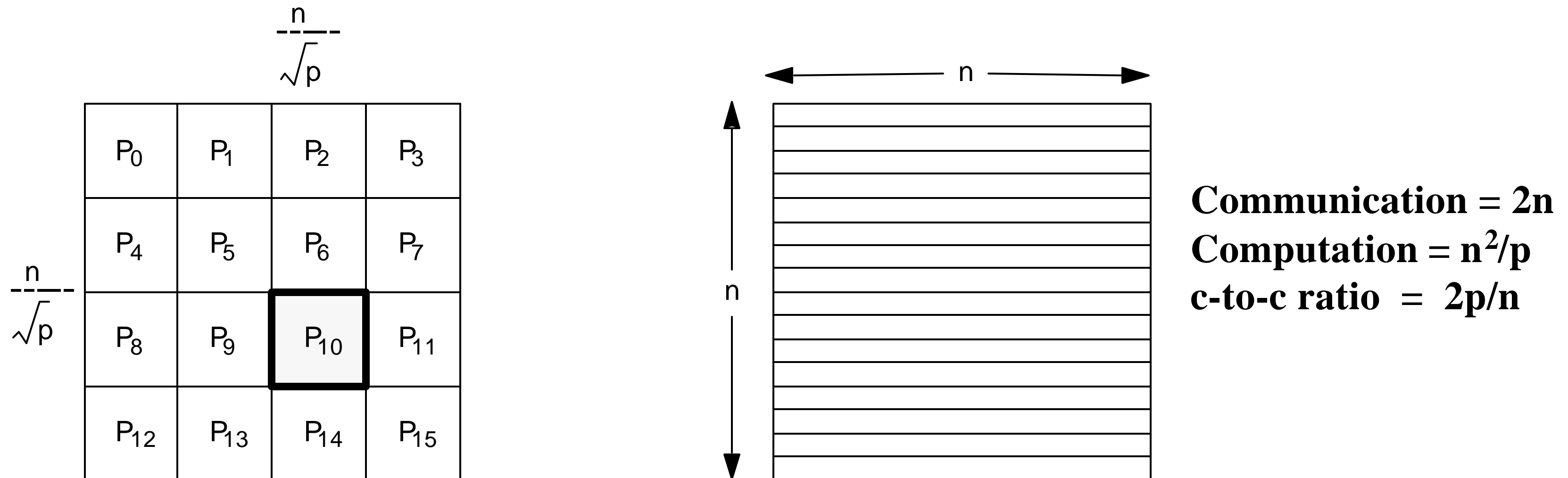
comm-to-comp ratio = Perimeter to Area (area to volume in 3-d)

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

Domain Decomposition (continued)

Best domain decomposition depends on information requirements

Nearest neighbor example: block versus strip decomposition:



Comm-to-comp ratio: $\frac{4 \times \sqrt{p}}{n}$ for block, $\frac{2 \times p}{n}$ for strip

Application dependent: strip may be better in other cases

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

Implications of Communication-to-Computation Ratio

- Architects must examine application latency/bandwidth needs
- If denominator in c-to-c is computation execution time, ratio gives average BW needs per task.
- If 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 and cost as well:**

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

$\rightarrow$ **Need to keep communication balanced across processors as well.**

Reducing Extra Work (Overheads)

- **Common sources of extra work (mainly orchestration):**
  - **Computing a good partition (at run time):**

    e.g. partitioning in Barnes-Hut or sparse matrix
  - **Using redundant computation to avoid communication.**
  - **Task, data distribution 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**

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

Summary of Parallel Algorithms Analysis

- **Requires characterization of multiprocessor system and algorithm requirements.**
- **Historical focus on algorithmic aspects: partitioning, mapping**
- **PRAM model: data access and communication are free**
  - **Only load balance (including serialization) and extra work matter**

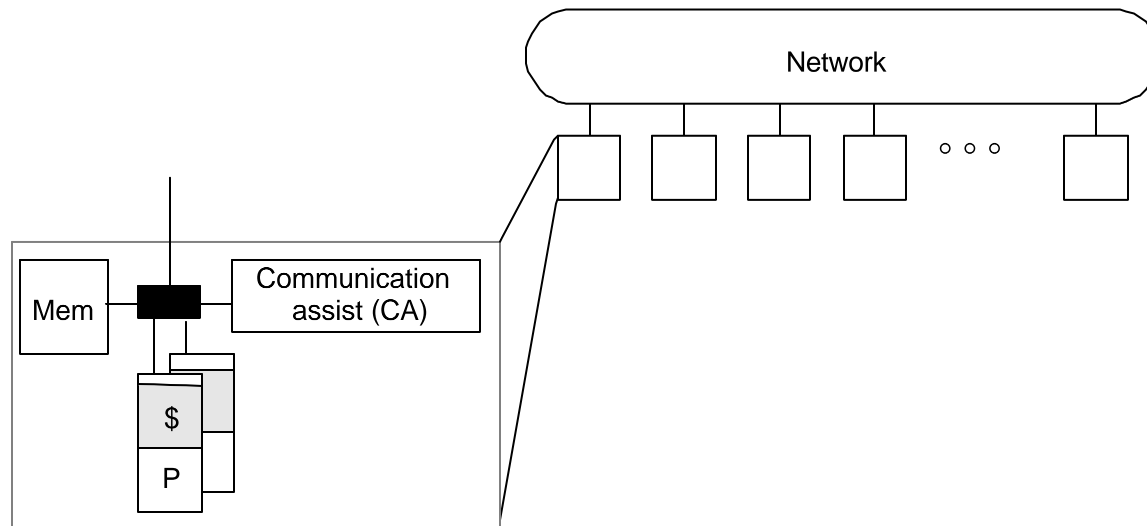
$$\text{Speedup} \leq \frac{\text{Sequential Instructions}}{\mathit{Max}\ (\text{Instructions} + \text{Synch Wait Time} + \text{Extra Instructions})}$$

  - **Useful for early development, but unrealistic for real performance**
  - **Ignores communication and also the imbalances it causes**
  - **Can lead to poor choice of partitions as well as orchestration**

Limitations of Parallel Algorithm Analysis

- Inherent communication in a parallel algorithm is not the only communication present:
  - Artifactual communication caused by program implementation and architectural interactions can even dominate.
  - Thus, actual amount of communication may not be 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.

Generic Multiprocessor Architecture



Node: processor(s), memory system, plus *communication assist:*

- **Network interface and communication controller.**

- **Scalable network.**

Extended Memory-Hierarchy View of Multiprocessors

- 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
  - 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

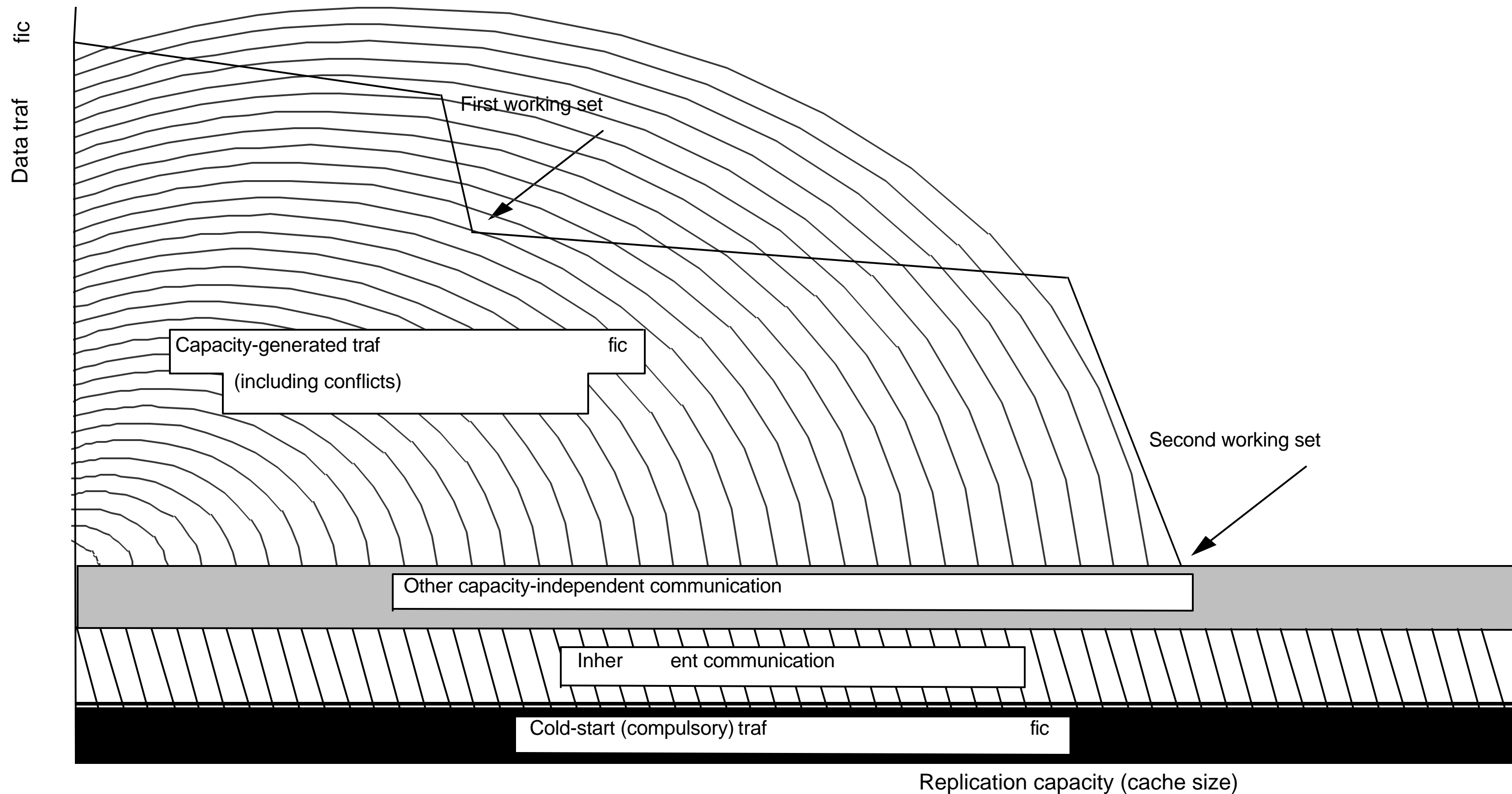
- Inherent communication, implicit or explicit, causes transfers:
  - Determined by program
- Artifactual communication:
  - Determined by program implementation and arch. interactions
  - Poor allocation of data across distributed memories: data accessed heavily by one node is located in another node local memory.
  - Unnecessary data in a transfer: More data communicated in a message than needed.
  - Unnecessary transfers due to system granularities (cache block size, page size).
  - Redundant communication of data: data value may change often but only last value needed.
  - Finite replication capacity (in cache or main memory)
- Inherent communication assumes unlimited capacity, small transfers, perfect knowledge of what is needed.
- More on artifactual communication later; first consider replication-induced further

Communication and Replication

- Comm. induced by finite capacity is most fundamental artifact
  - Similar to cache size and miss rate or memory traffic in uniprocessors.
  - Extended memory hierarchy view useful for this relationship
- View as three level hierarchy for simplicity
  - Local cache, local memory, remote memory (ignore network topology).
- Classify "misses" in "cache" at any level as for uniprocessors
    - *Compulsory* or *cold* misses (no size effect)
    - *Capacity* misses (yes)
    - *Conflict* or *collision* misses (yes)
    - *Communication* or *coherence* misses (no)
  - Each may be helped/hurt by large transfer granularity (spatial locality).

Working Set Perspective

- At a given level of the hierarchy (to the next further one)



- **Hierarchy of working sets**
- **At first level cache (fully assoc, one-word block), inherent to algorithm**
  - ***working set curve*** **for program**
- **Traffic from any type of miss can be local or nonlocal (communication)**

Orchestration for Performance

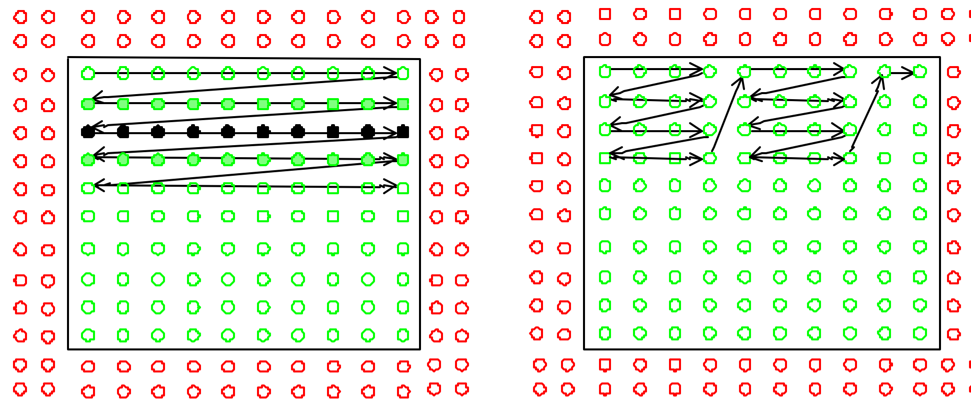
- Reducing amount of communication:
  - Inherent: change logical data sharing patterns in algorithm
  - Artifactual: exploit spatial, temporal locality in extended hierarchy
    - Techniques often similar to those on uniprocessors
- Structuring communication to reduce cost
- We'll examine techniques 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 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)

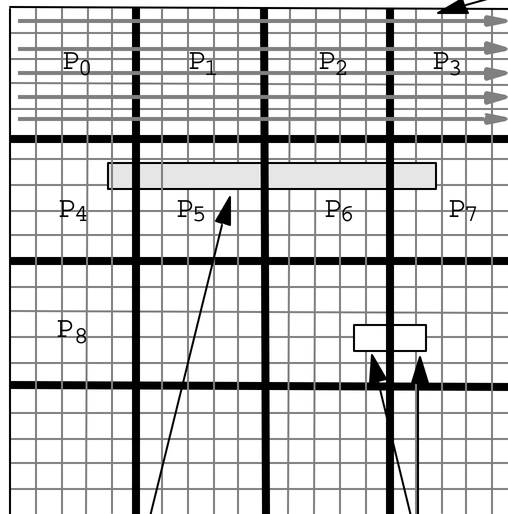
Exploiting Spatial Locality

- 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

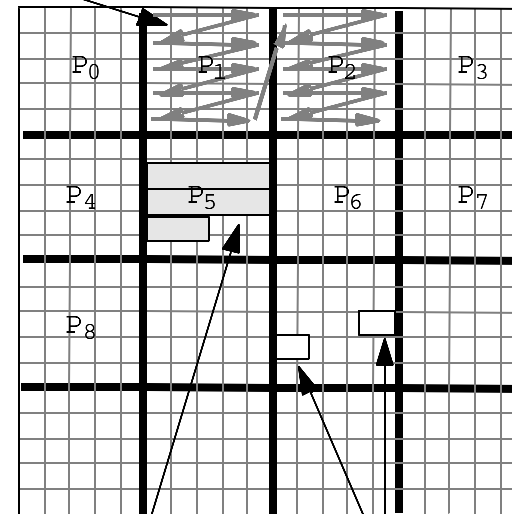
Spatial Locality Example

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

Contiguity in memory layout



(a) Two-dimensional array

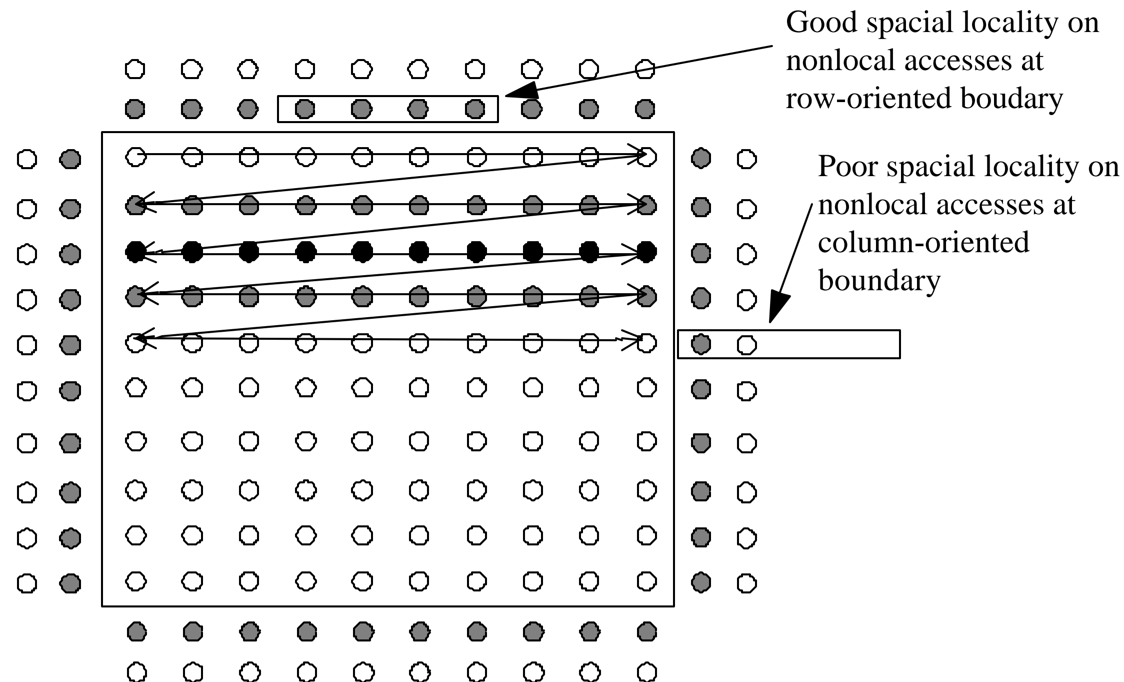


(b) Four-dimensional array

Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows

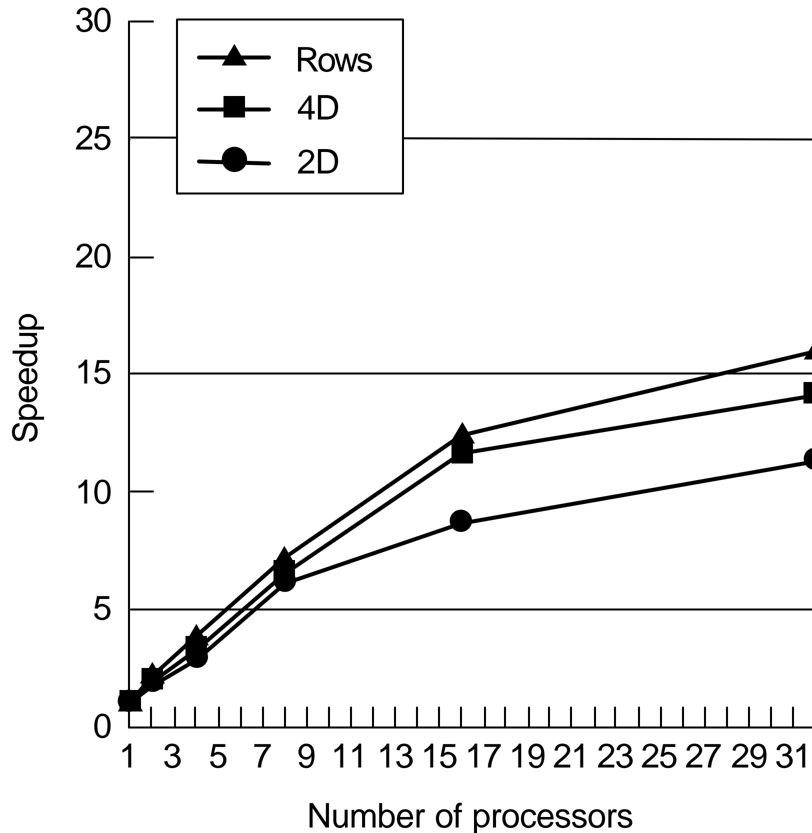
- **Blocks still have a spatial locality problem on remote data**
- **Row-wise can perform better despite worse inherent c-to-c ratio**



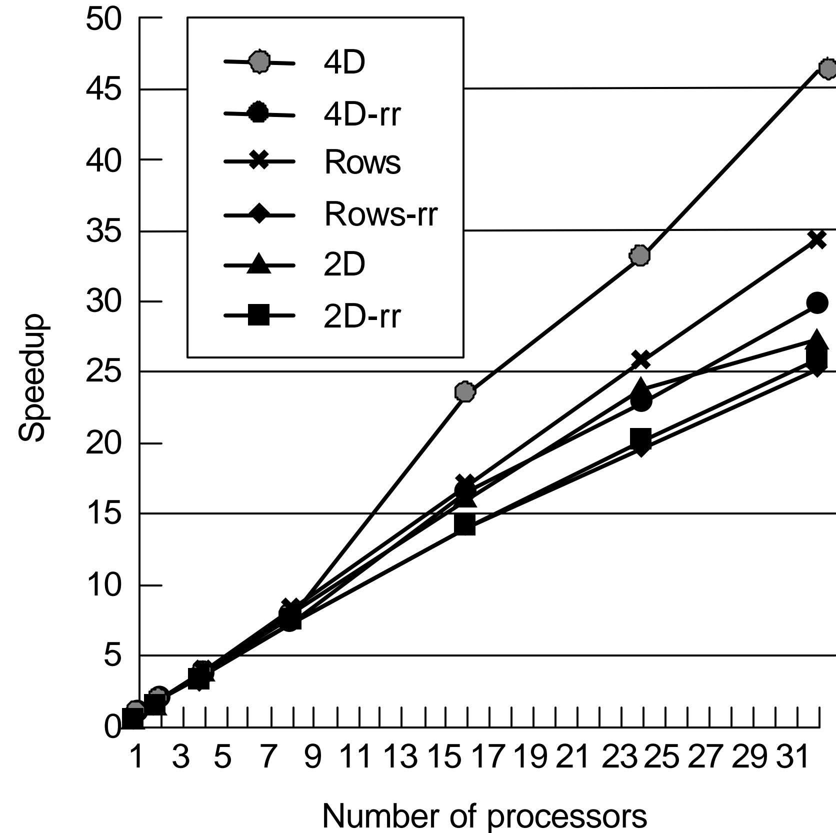
- Result depends on *n* and *p*

Example Performance Impact

Equation solver on SGI Origin2000



514 x 514 grids



12k x 12k grids

Structuring Communication

Given amount of comm (inherent or artifactual), 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 contention 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 no. of messages *m* or overhead per message *o***
- ***o* is usually determined by hardware or system software**
  - **Program should try to reduce m by combining messages.**
  - **More control when communication is explicit (message-passing).**
- **Combine 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**
      - **Cobmining 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

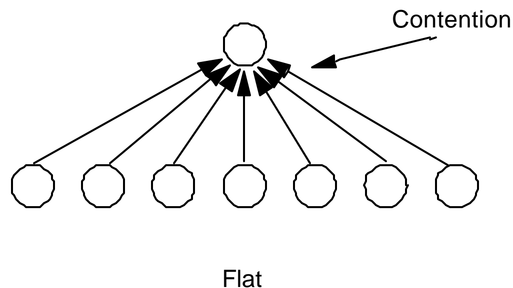
    e.g. nearest-neighbor on mesh and ring; all-to-all
  - How important is this?
    - Used to be a major focus of parallel algorithms
    - Depends on no. of processors, how t_h, compares with other components
    - Less important on modern machines

Reducing Contention

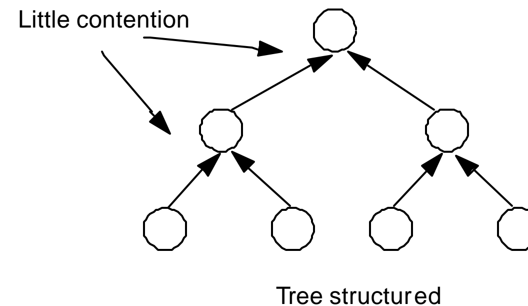
- All resources have nonzero occupancy (busy time):
  - Memory, communication controller, network link, etc. Can only handle so many transactions per unit time.
  - Results in queuing delays at the busy resource.
- Effects of contention:
  - Increased end-to-end cost for messages.
  - Reduced available 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



Flat



Tree structured

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

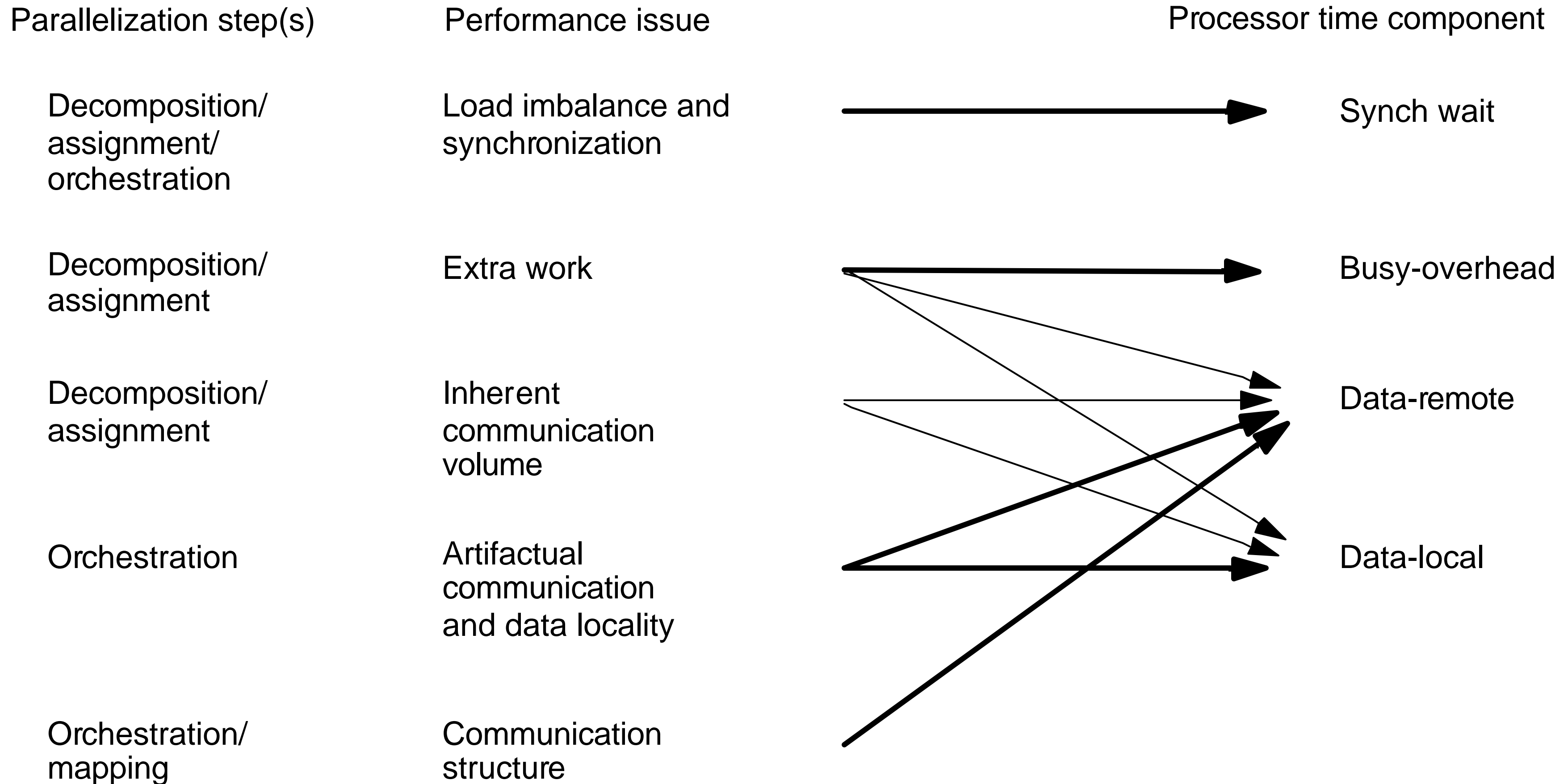
Overlapping Communication

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

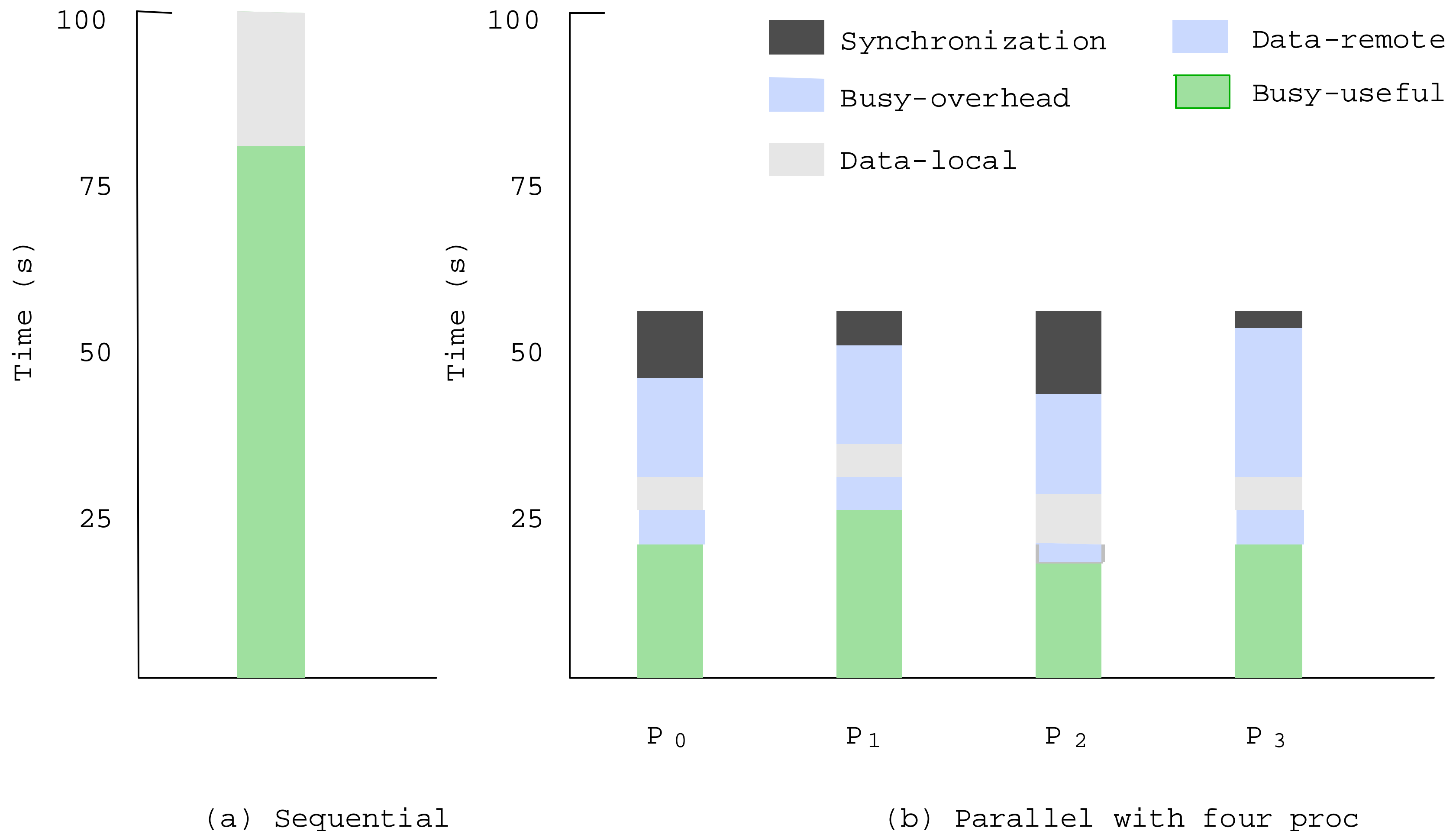
Summary of Tradeoffs

- 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



Components of Execution Time From Processor Perspective



Summary

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

- 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