ECE 669

Parallel Computer Architecture

Lecture 6

Programming for Performance

Introduction

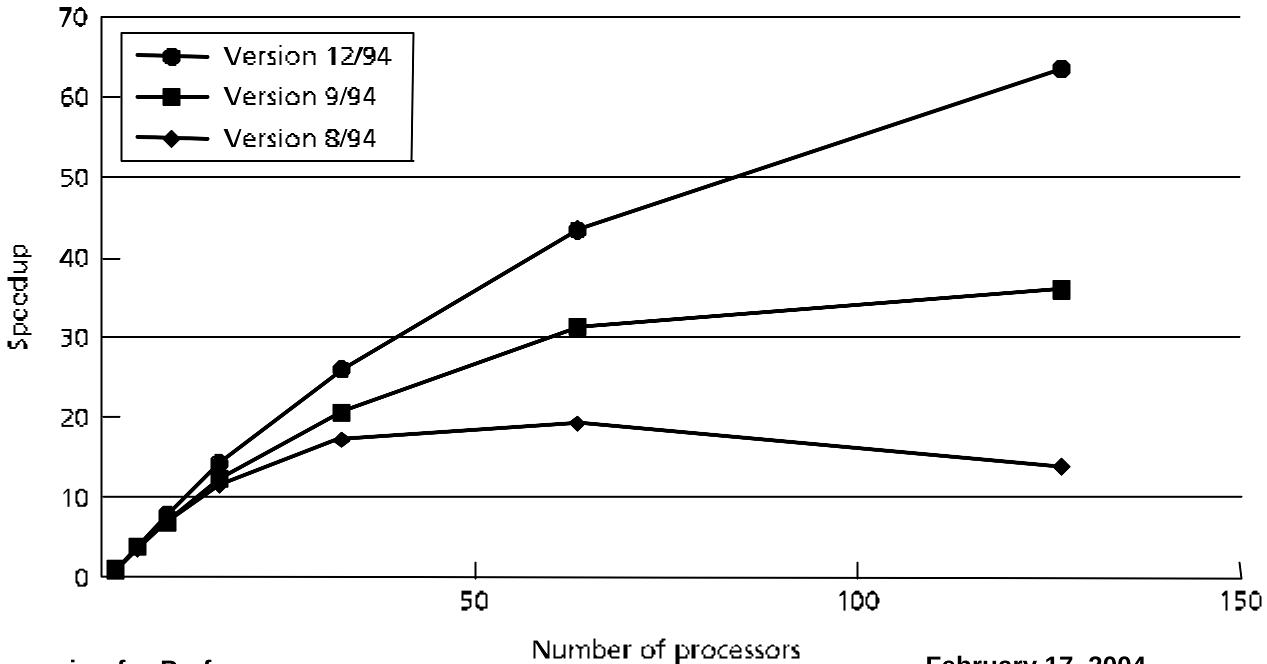
- Rich space of techniques and issues
  - Trade off and interact with one another
- Issues can be addressed/helped by software or hardware
  - Algorithmic or programming techniques
  - Architectural techniques
- Focus here on performance issues and software techniques
  - Partitioning
  - Communication
  - Orchestration

Partitioning for Performance

- Initially consider how to segment program without view of programming model
- Important factors:
  - Balancing workload
  - Reducing communication
  - Reducing extra work needed for management
- Goals similar for parallel computer and VLSI design
- Algorithms or manual approaches
- Perhaps most important factor for performance

Performance Goal => Speedup

- **Architect Goal**
  - **observe how program uses machine and improve the design to enhance performance**
- **Programmer Goal**
  - **observe how the program uses the machine and improve the implementation to enhance performance**
- **What do you observe?**
- **Who fixes what?**



Partitioning for Performance

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

- Even these algorithmic issues trade off:
  - Minimize comm. => 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
- Goal is to compromise
  - Fortunately, often not difficult in practice

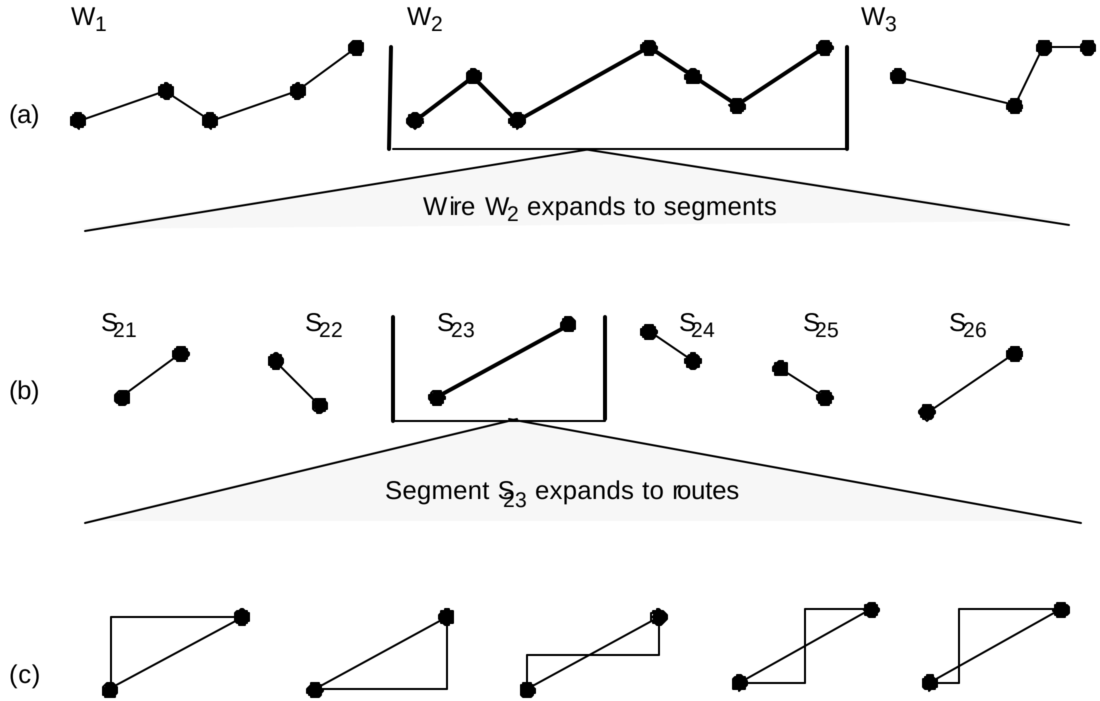
Load Balance and Synch Wait Time

- **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**

- **Four parts to load balance 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**

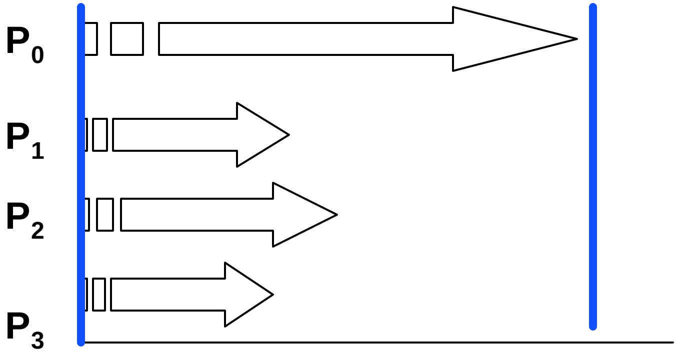
Identifying Concurrency

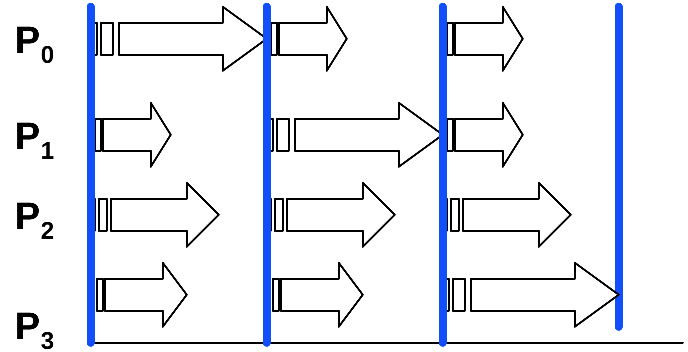
- **Techniques seen for equation solver:**
  - **Loop structure, fundamental dependences, new algorithms**
- ***Data Parallelism* versus *Function Parallelism***
- **Often see orthogonal levels of parallelism; e.g. VLSI routing**



Load Balance and Synchronization

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





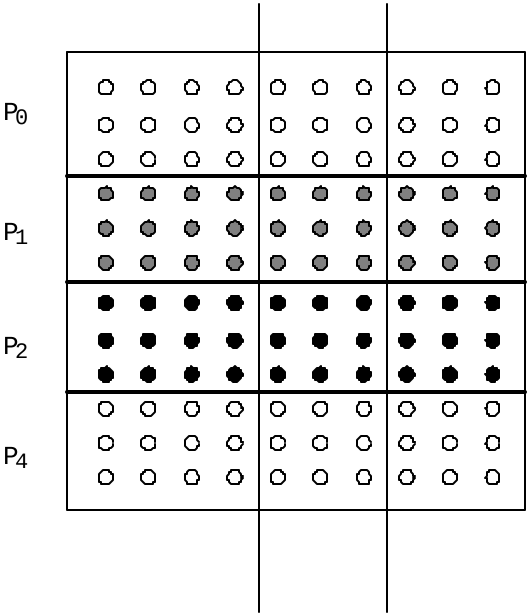
- **Instantaneous load imbalance revealed as wait time**
  - **at completion**
  - **at barriers**
  - **at receive**

$$\frac{\text{Sequential Work}}{\textit{Max}\ (\text{Work} + \text{Synch Wait Time})}$$

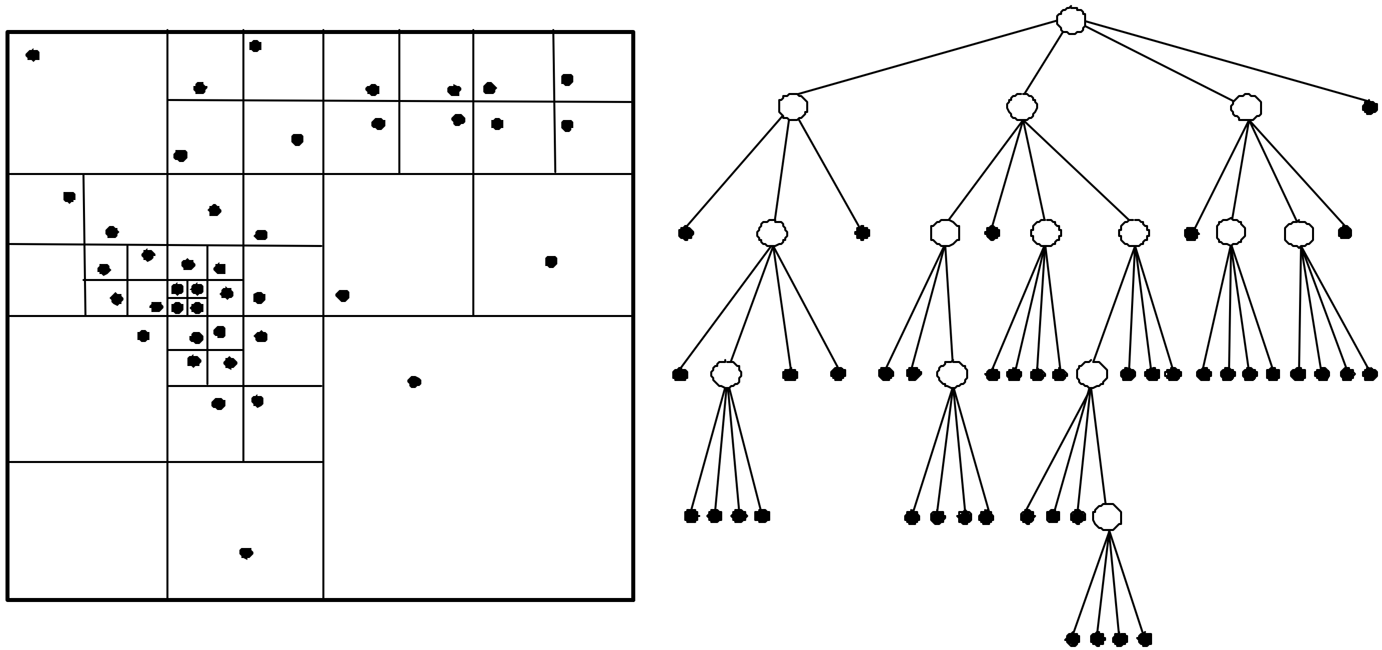
Improving Load Balance

- Decompose into more smaller tasks (>>P)
- Distribute uniformly
  - variable sized task
  - randomize
  - bin packing
  - dynamic assignment
- Schedule more carefully
  - avoid serialization
  - estimate work
  - use history info.

```
for_all i = 1 to n do
    for_all j = i to n do
        A[ i, j ] = A[i-1, j] + A[i, j-1] + ...
```



Example: Barnes-Hut

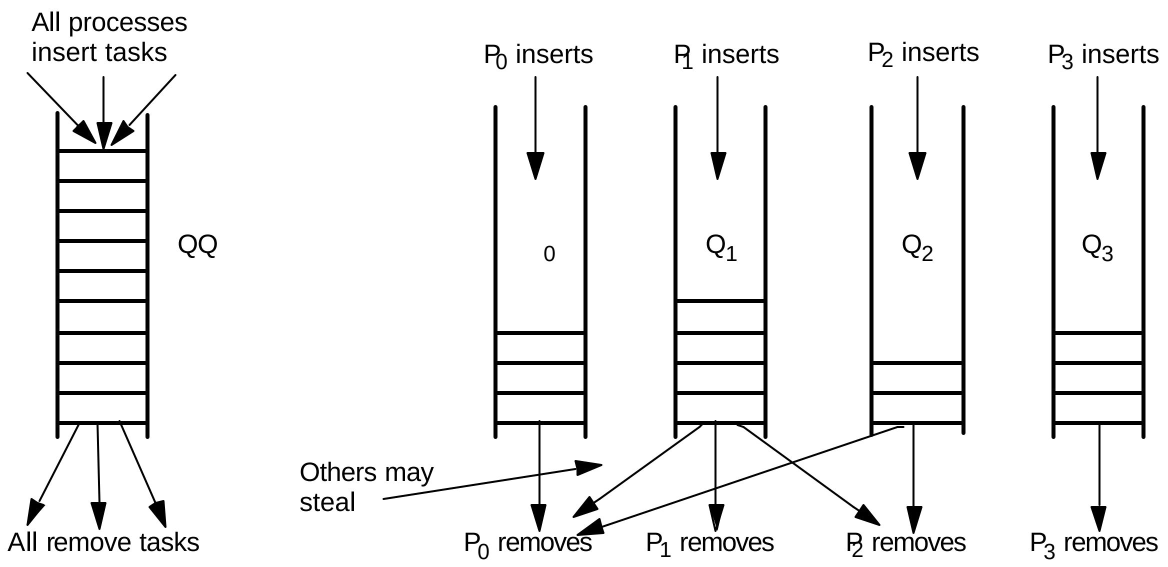


(a) The spatial domain (b) Quadtr ee r epr esentation

- **Divide space into roughly equal # particles**
- **Particles close together in space should be on same processor**
- **Nonuniform, dynamically changing**

Dynamic Scheduling with Task Queues

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



(a) Centralized task queue (b) Distributed task queues (one per pr ocess)

Deciding How to Manage Concurrency

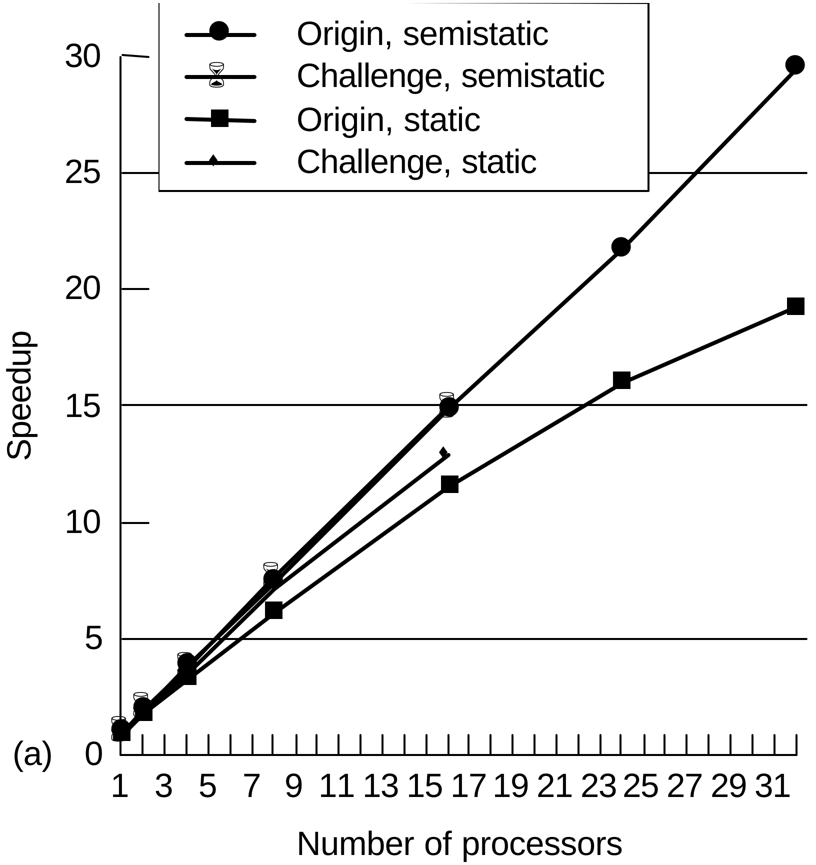
- ***Static*** **versus** ***Dynamic*** **techniques**
- **Static:**
  - **Algorithmic assignment based on input; won't change**
  - **Low runtime overhead**
  - **Computation must be predictable**
  - **Preferable when applicable (except in multiprogrammed/heterogeneous environment)**
- **Dynamic:**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

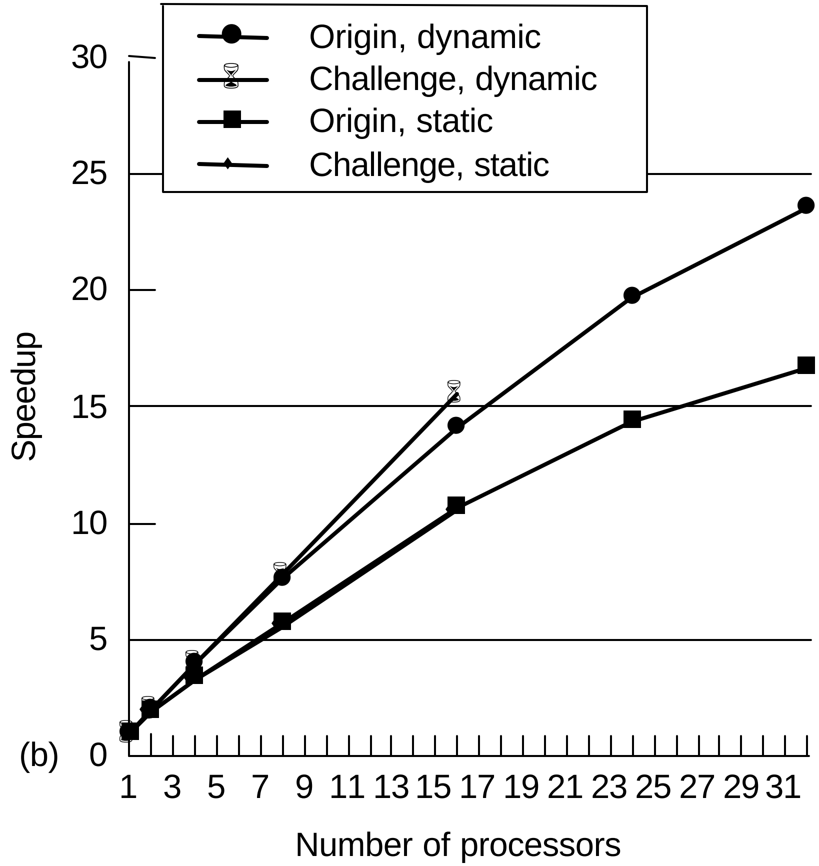
Dynamic Assignment

- **Profile-based (semi-static):**
  - **Profile work distribution at runtime, and repartition dynamically**
  - **Applicable in many computations, e.g. Barnes-Hut, 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)**

Impact of Dynamic Assignment

- **Barnes-Hut and Ray Tracing on SGI Origin 2000 and Challenge (cache-coherent shared memory)**
- **Semistatic – periodic run-time re-evaluation of task assignment**





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 and contention

- Communication and contention affected by assignment, not size
  - Overhead an issue, particularly with task queues

Reducing Serialization

- **Be careful about assignment and orchestration**
  - including scheduling
- **Event synchronization**
  - Reduce use of conservative synchronization
    - Point-to-point instead of global barriers
  - 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, more space, less reuse
  - Smaller, less frequent critical sections
    - don't do reading/testing in critical section, only modification

Implications of Load Balance

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

- **Extends speedup limit expression to:**
- **Generally, 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 balance**
  - **N*aming* logically shared data in the presence of task stealing**
  - **Efficient support for point-to-point communication**

Architectural Implications of Load Balancing

- **Naming**
  - **Global position independent naming separates decomposition from layout**
  - **Allows diverse, even dynamic assignments**
- **Efficient fine-grained communication & synch**
  - **Requires:**
    - **messages**
    - **locks**
  - **point-to-point synchronization**
- **Automatic replication of tasks**

Implications of Comm-to-Comp Ratio

- **Architects examine application needs to see where to spend money**
- **If denominator is execution time, ratio gives average BW needs**
- **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 comm. depends on structure and cost as well**

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

  - **Need to keep communication balanced across processors as well**

Reducing Extra Work

- **Common sources of extra work:**
  - **Computing a good partition**
    - **e.g. partitioning in Barnes-Hut**
  - **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**

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

Reducing Inherent Communication

- **Communication is expensive!**
- **Measure: *communication to computation ratio***
- **Inherent communication**
  - **Determined by assignment of tasks to processes**
  - **One produces data consumed by others**
- **Replicate computations**

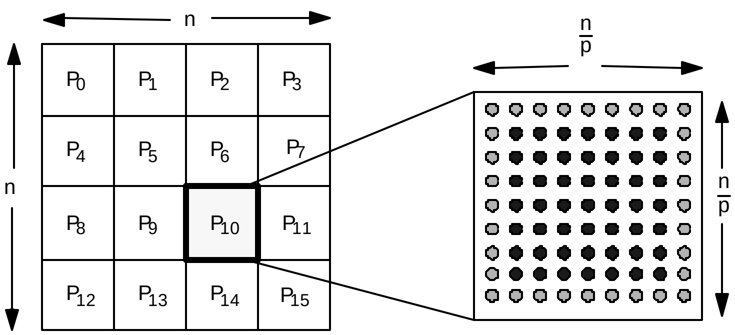
=> Use algorithms that communicate less

=> Assign tasks that access same data to same process

- **same row or block to same process in each iteration**

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



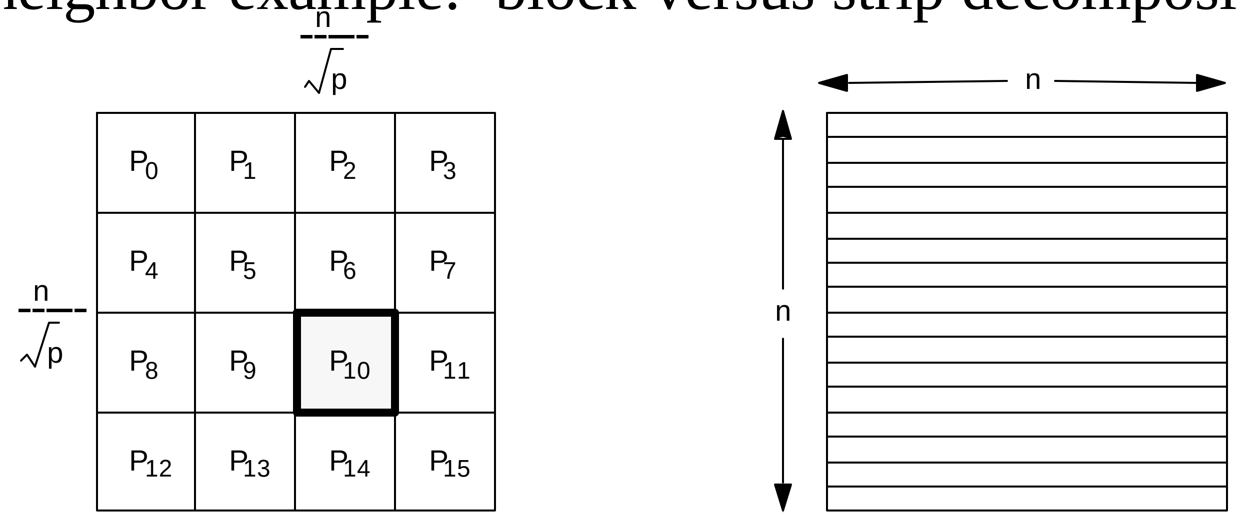
Perimeter to Area comm-to-comp ratio (area to volume in 3-d)

- 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:** $\frac{4*p^{0.5}}{n}$ **for block,** $\frac{2*p}{n}$ **for strip**
- **Application dependent: strip may be better in other cases**

Finding a Domain Decomposition

- **Static, by inspection**
  - Must be predictable: grid example, Ocean
- **Static, but not by inspection**
  - Input-dependent, require analyzing input structure
  - E.g sparse matrix computations
- **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

Summary: Analyzing Parallel Algorithms

- Requires characterization of multiprocessor and algorithm
- 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

Orchestration for Performance

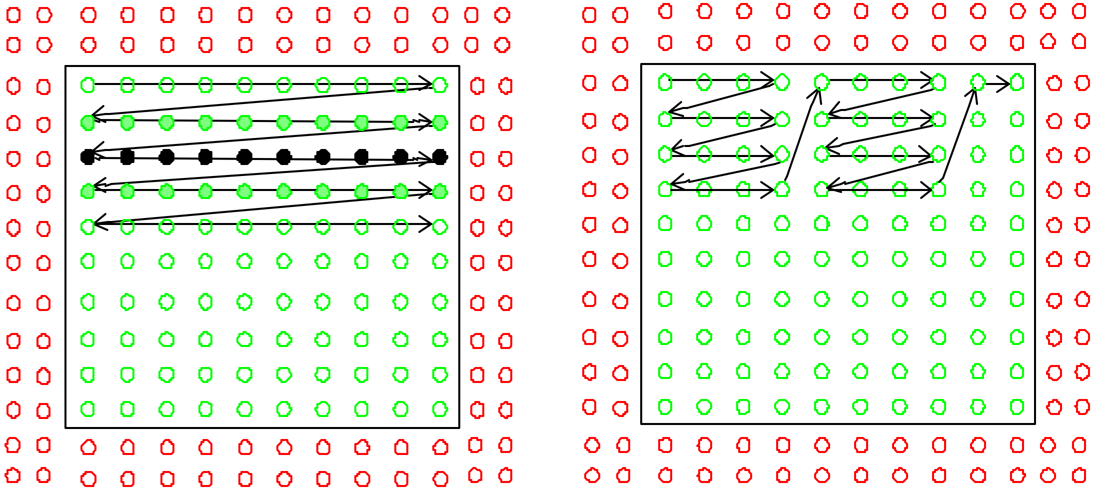
- **Reducing amount of communication:**
  - **Inherent: change logical data sharing patterns in algorithm**
  - **Artifactual: exploit spatial, temporal locality in extended hierarchy**
    - **Techniques often similar to those on uniprocessors**
- **Structuring communication to reduce cost**

Reducing Communication

- **Message passing model**
  - Communication and replication are both explicit
  - Communication is in 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**
- **Solver example: blocking**



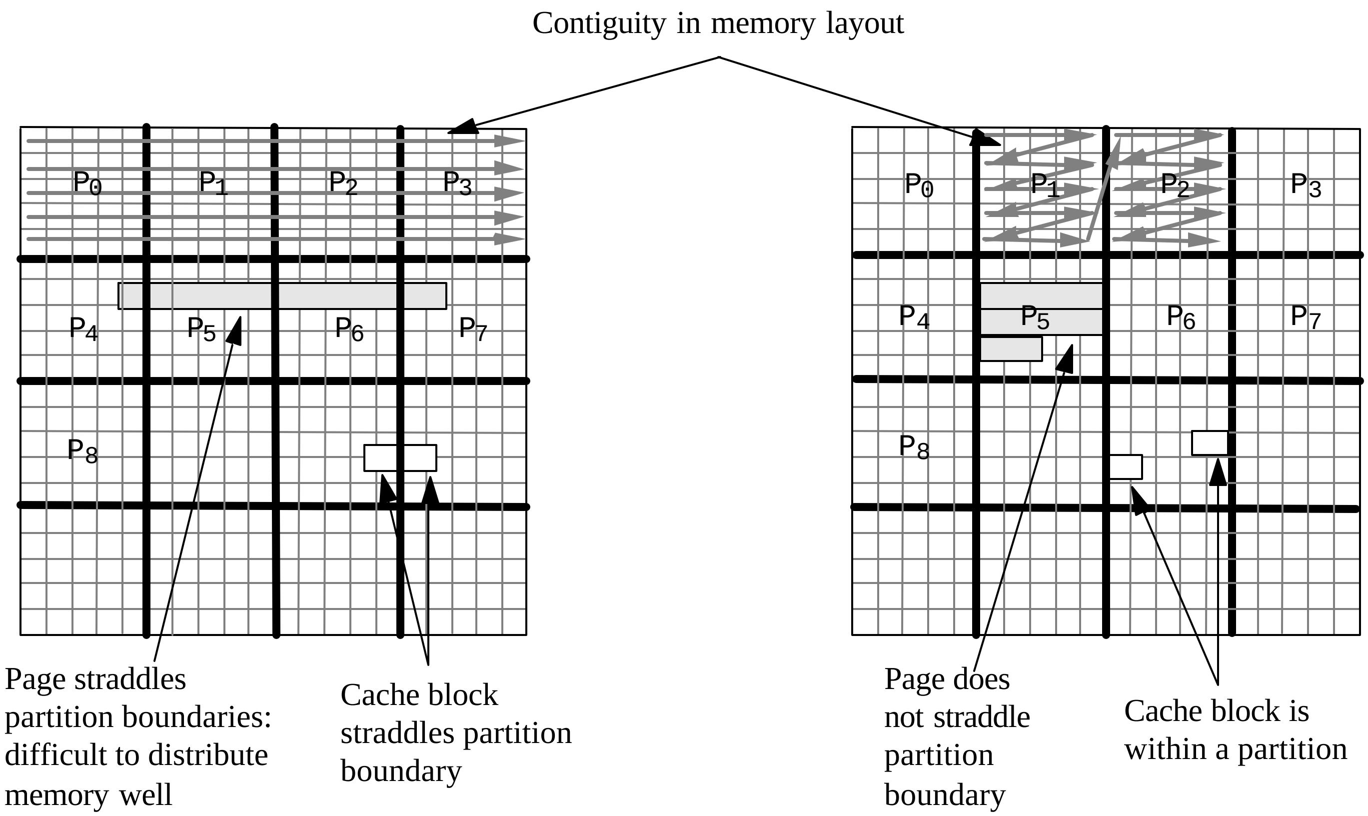
(a) Unblocked access pattern in a sweep (b) Blocked access pattern with B = 4

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



(a) Two-dimensional array

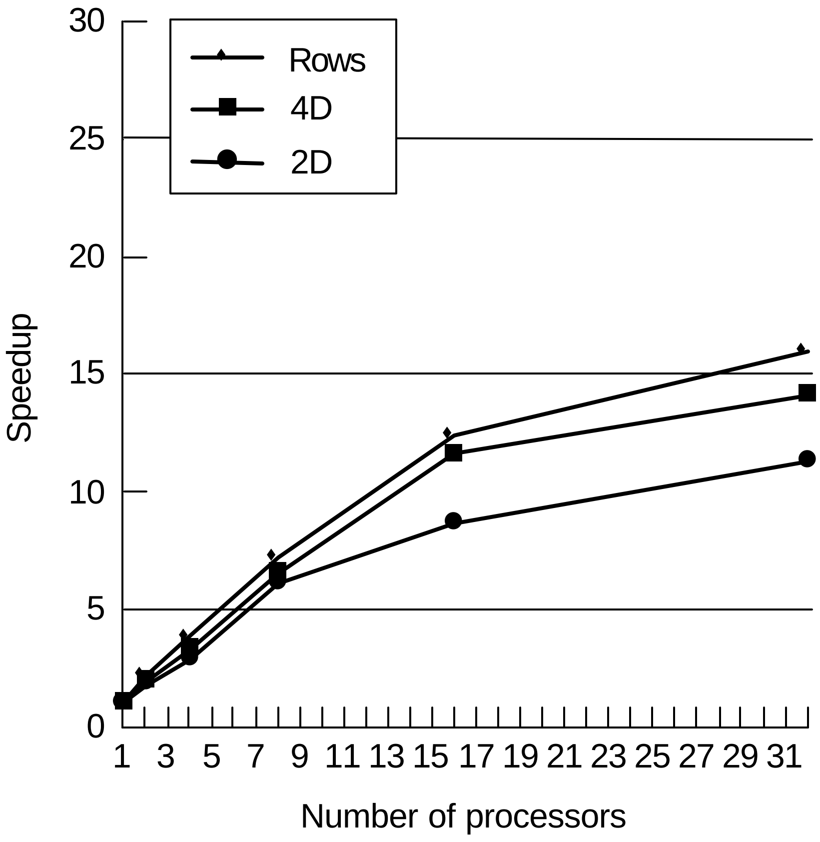
(b) Four-dimensional array

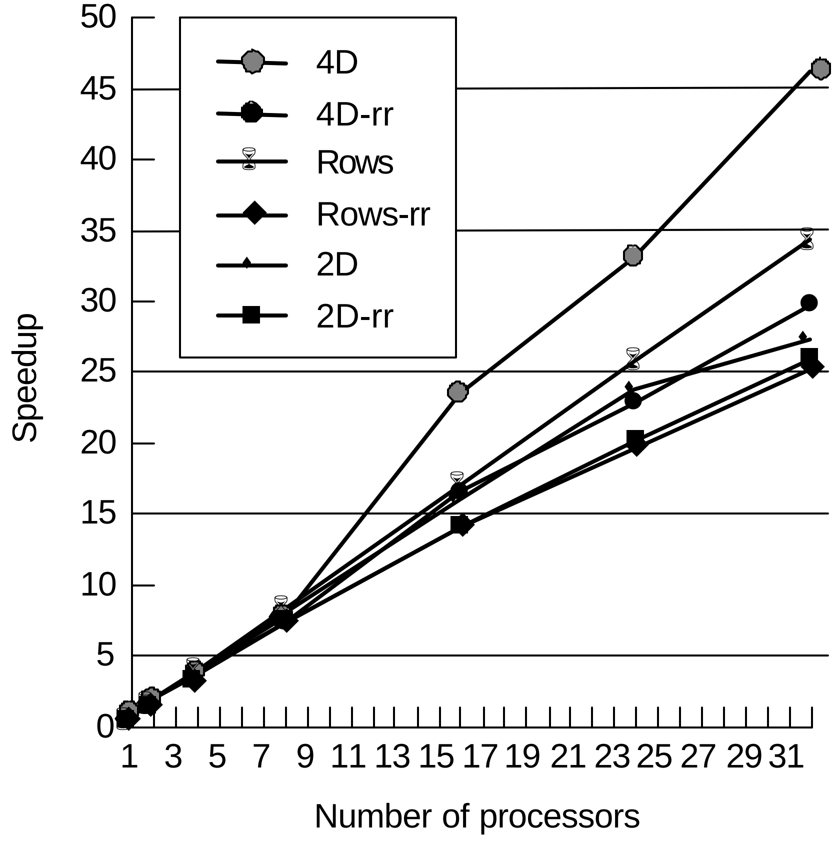
Architectual Implications of Locality

- Communication abstraction that makes exploiting it easy
- For cache-coherent SAS
  - Size and organization of levels of memory hierarchy
    - cost-effectiveness: caches are expensive
    - caveats: flexibility for different and time-shared workloads
  - Replication in main memory useful? If so, how to manage?
    - hardware, OS/runtime, program?
  - Granularities of allocation, communication, coherence (?)
    - small granularities => high overheads, but easier to program
- Machine granularity (resource division among processors, memory...)

Example Performance Impact

- **Equation solver on SGI Origin2000**





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**