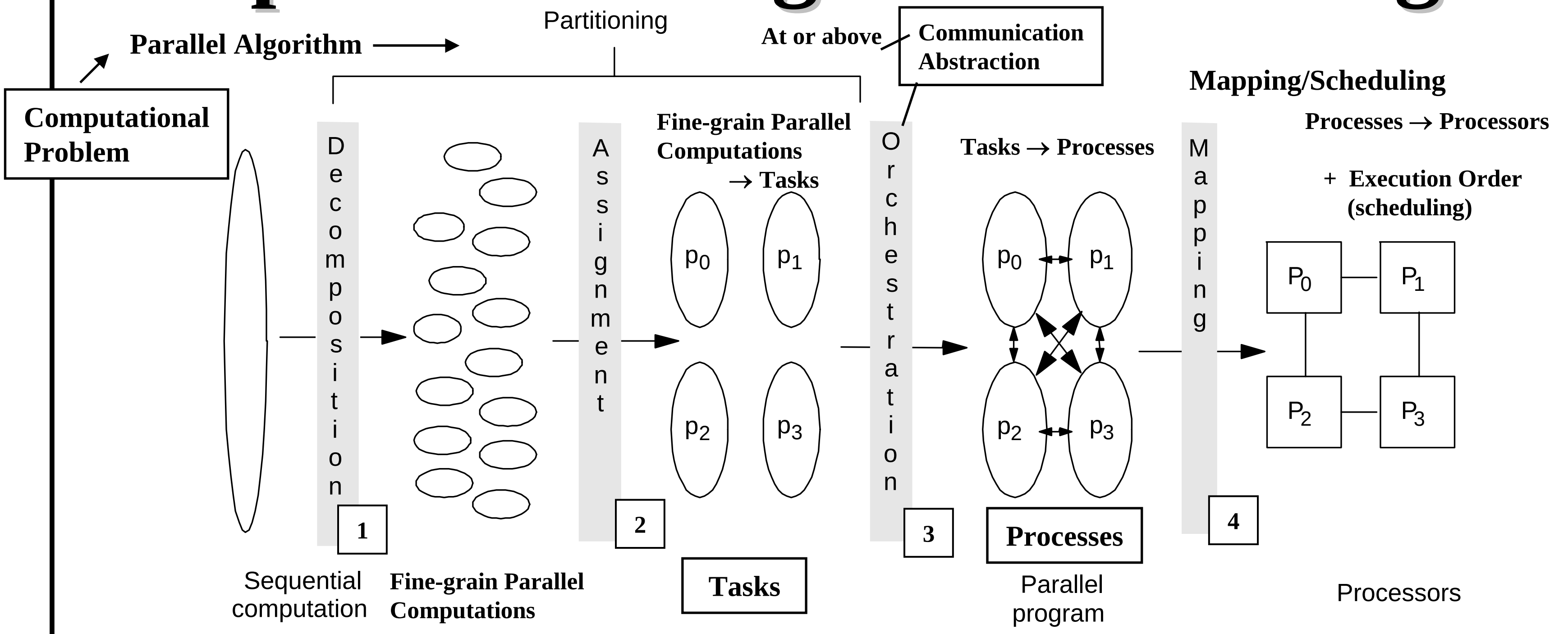
Steps in Creating a Parallel Program



Parallel Algorithm

Partitioning

At or above

Communication Abstraction

Mapping/Scheduling

Computational Problem

Decomposition

Assignment

Fine-grain Parallel Computations → Tasks

Orchestration

Tasks → Processes

Mapping

Processes → Processors

+ Execution Order (scheduling)

p_0 p_1 p_2 p_3

p_0 p_1 p_2 p_3

P_0 P_1 P_2 P_3

1 2 3 4

Sequential computation

Fine-grain Parallel Computations

Tasks

Processes

Parallel program

Processors

- **4 steps: Decomposition, Assignment, Orchestration, Mapping + Scheduling**
- **Performance Goal of the steps: Maximize parallel speedup) (minimize resulting parallel) execution time by:**
  1. **<u>Balancing computations and overheads</u> on processors (every processor does the same amount of work + overheads).**
  2. **<u>Minimizing communication cost and other overheads</u> associated with each step.**

(Parallel Computer Architecture, Chapter 3)

Parallel Programming for Performance

A process of Successive Refinement of the steps

- **Partitioning for Performance:**
  - **Load Balancing and Synchronization Wait Time Reduction**
  - **Identifying & Managing Concurrency**
    - **Static Vs. Dynamic Assignment**
    - **Determining Optimal Task Granularity**
    - **Reducing Serialization / Synch Wait Time**
  - **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**

or tasking

Waiting time as a result of data dependency

C-to-C Ratio

or “Extra”

(Parallel Computer Architecture, Chapter 3)

Successive Refinement of Parallel Program Performance

Partitioning is possibly independent of architecture, and may be done first (initial partition):

- View machine as a collection of communicating processors
  - Balancing the workload across tasks/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:
  - Reduce artifactual (extra) communication due to architectural interactions.
  - Cost of communication depends on how it is structured (possible overlap with computation) + Hardware Architecture
- This may inspire changes in partitioning.

And algorithm?

Partitioning for Performance

- 1 **<u>Balancing the workload across tasks/processes:</u>**
  - **Reducing wait time at synchronization points needed to satisfy data dependencies among tasks.**
- 2 **<u>Reduce Overheads:</u>**
  - **<u>Reducing interprocess inherent communication.</u>**
  - **<u>Reducing extra work needed to find a good assignment.</u>**

The above goals lead to two extreme trade-offs:

- **Minimize communication => run on 1 processor.** **One large task**

  ? ↕ **=> extreme load imbalance.**
- **Maximize load balance => random assignment of <u>tiny</u> tasks.**

  => no control over communication.
- **A good partition may imply extra work to compute or manage it**

- **The goal is to <u>compromise</u> between the above extremes**

Load Balancing and Synch Wait Time Reduction

Synch wait time = process/task wait time as a result of data dependency on another task (until the dependency is satisfied)

Limit on speedup:

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

(on any processor)

- **Work includes computing, data access and other costs.**
- **Not just equal work, but must be busy (computing) at same time to minimize synchronization wait time to satisfy dependencies.**

Four parts to load balancing and reducing synch wait time:

1. **Identify enough concurrency in decomposition.**
2. **Decide how to manage the concurrency (statically or dynamically).**
3. **Determine the granularity (task grain size) at which to exploit it.**
4. **Reduce serialization and cost of synchronization.**

Identifying Concurrency: Decomposition

- **Concurrency may be found by:**
  - 1 – **Examining loop structure of sequential algorithm.**
  - 2 – **Fundamental data dependencies (dependency analysis/graph).**
  - 3 – **Exploit the understanding of the problem to devise parallel algorithms with more concurrency (e.g ocean equation solver).**

- **Software/Algorithm Parallelism Types:**

1 - Data Parallelism versus 2- Functional Parallelism:

1 - 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.

Software/Algorithm Parallelism Types: were also covered in lecture 3 slide 33

Identifying Concurrency (continued)

2- Functional Parallelism:

- **Entire large tasks (procedures) with possibly different functionality 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.**
- **Concurrency degree usually modest and does not grow with input size**
  - **Difficult to load balance.**
  - **Often used to reduce synch wait time between data parallel phases.**

Most scalable parallel programs:

(more concurrency as problem size increases) parallel programs:

Data parallel programs (per this loose definition)

- **Functional parallelism can still be exploited to reduce synchronization wait time between data parallel phases.**

Software/Algorithm Parallelism Types: were also covered in lecture 3 slide 33

Managing Concurrency: Assignment

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

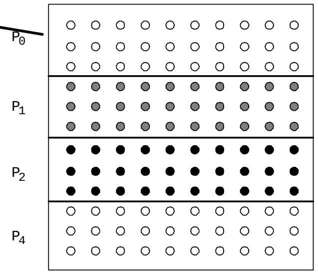
***Static* versus *Dynamic* Assignment:**

Static Assignment: (e.g equation solver)

At Compilation Time

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

of computations into tasks at compilation time



Example 2D Ocean Equation Solver

Dynamic Assignment: Or tasking

At Run Time

- **Needed when computation not fully predictable.**
- **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.**

Counts as extra work

Dynamic Assignment/Mapping

Profile-based (semi-static):

Initial partition

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

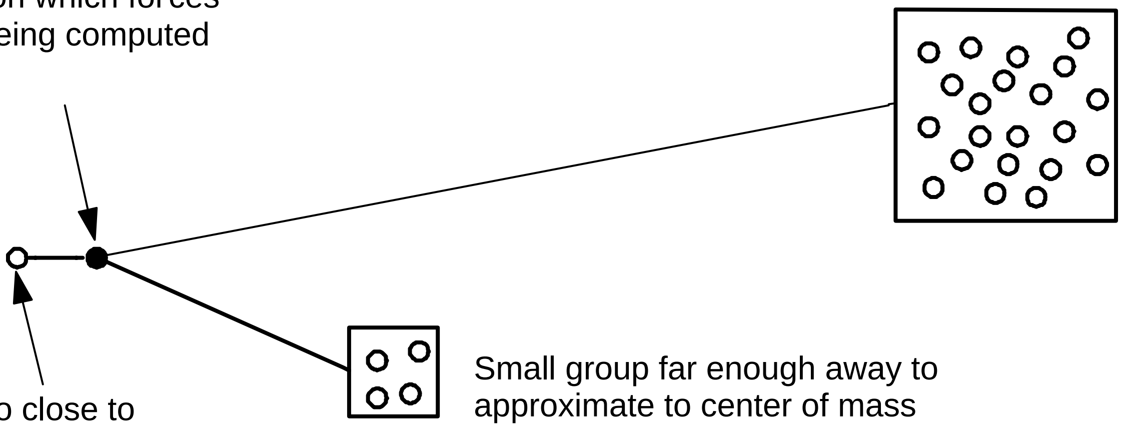
Dynamic Tasking:

- Deal with unpredictability in program or environment (e.g. Ray tracing)
  - Computation, communication, and memory system interactions
  - Multiprogramming and heterogeneity of processors
  - Used by runtime systems and OS too.
- Pool (queue) of tasks: Processors take and add tasks to pool until parallel computation is done.
- e.g. "self-scheduling" of loop iterations (shared loop counter).

Simulating Galaxy Evolution

(Gravitational N-Body Problem)

- **Simulate the interactions of many stars evolving over time**
- **Computing forces is expensive**
  - **$O(n^2)$ brute force approach**
- **Hierarchical Methods (e.g. Barnes-Hut) take advantage of force law: *G* *(center of mass)*** $G\frac{m_1 m_2}{r^2}$



Star on which forces are being computed

Large group far enough away to approximate **(using center of gravity)**

Star too close to approximate

Small group far enough away to approximate to center of mass

- Many time-steps, plenty of concurrency across stars within one

Gravitational N-Body Problem:
Barnes-Hut Algorithm

- **To parallelize problem: Groups of bodies partitioned among processors. Forces communicated by messages between processors.**
  - **Large number of messages, O(N^2) for one iteration.**
- **<u>Solution: Approximate a cluster of distant bodies as one body with their total mass</u>**
- **This clustering process can be applies recursively.**
- **Barnes_Hut: Uses divide-and-conquer clustering. For 3 dimensions:**
  - **Initially, one cube contains all bodies**
  - **Divide into 8 sub-cubes. (4 parts in two dimensional case).**
  - **If a sub-cube has no bodies, delete it from further consideration.**
  - **If a cube contains more than one body, recursively divide until each cube has one body**
  - **This creates an oct-tree which is very unbalanced in general.**
  - **After the tree has been constructed, the total mass and center of gravity is stored in each cube.**
  - **The force on each body is found by traversing the tree starting at the root stopping at a node when clustering can be used.**
  - **The criterion when to invoke clustering in a cube of size d x d x d:**

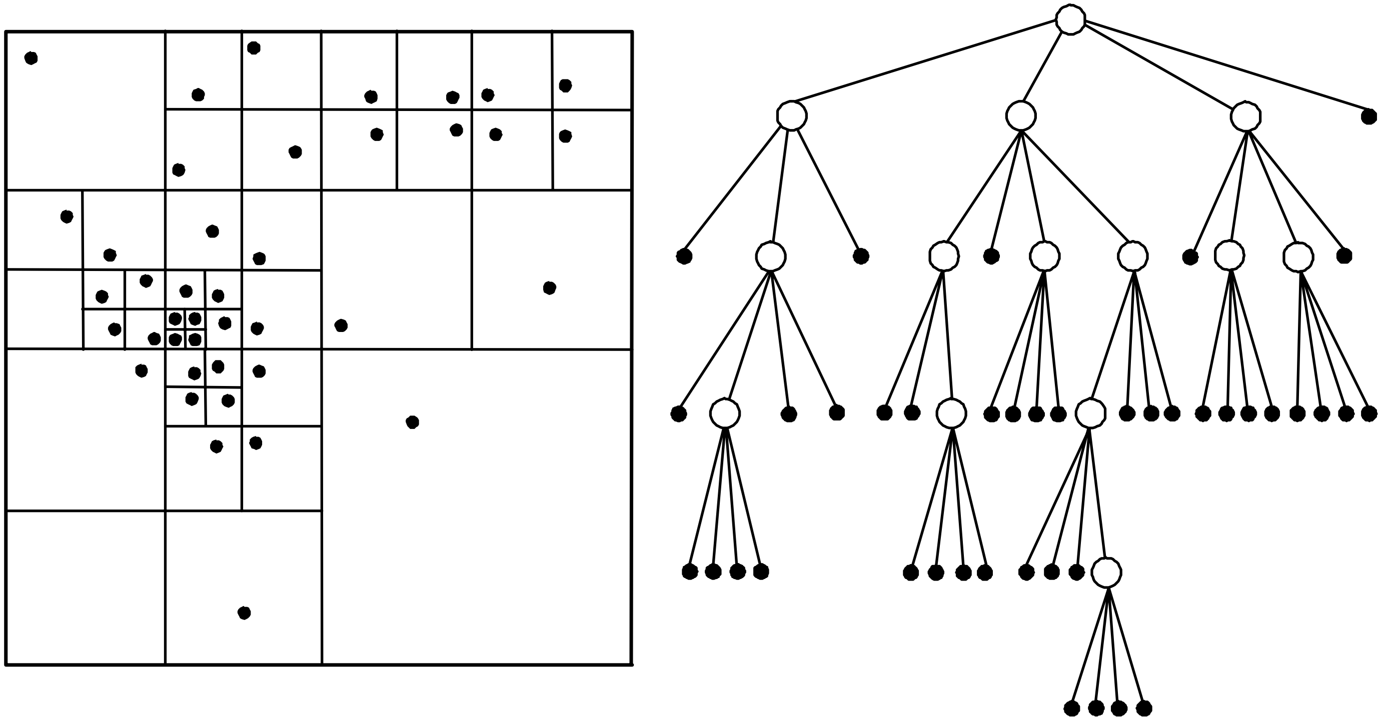
$$r \geq d/\theta$$

r = distance to the center of mass

θ = a constant, 1.0 or less, opening angle

  - **Once the new positions and velocities of all bodies is computed, the <u>process is repeated</u> for each time period requiring the oct-tree to be reconstructed (<u>repartition dynamically)</u>**

Two-Dimensional Barnes-Hut



(a) The spatial domain

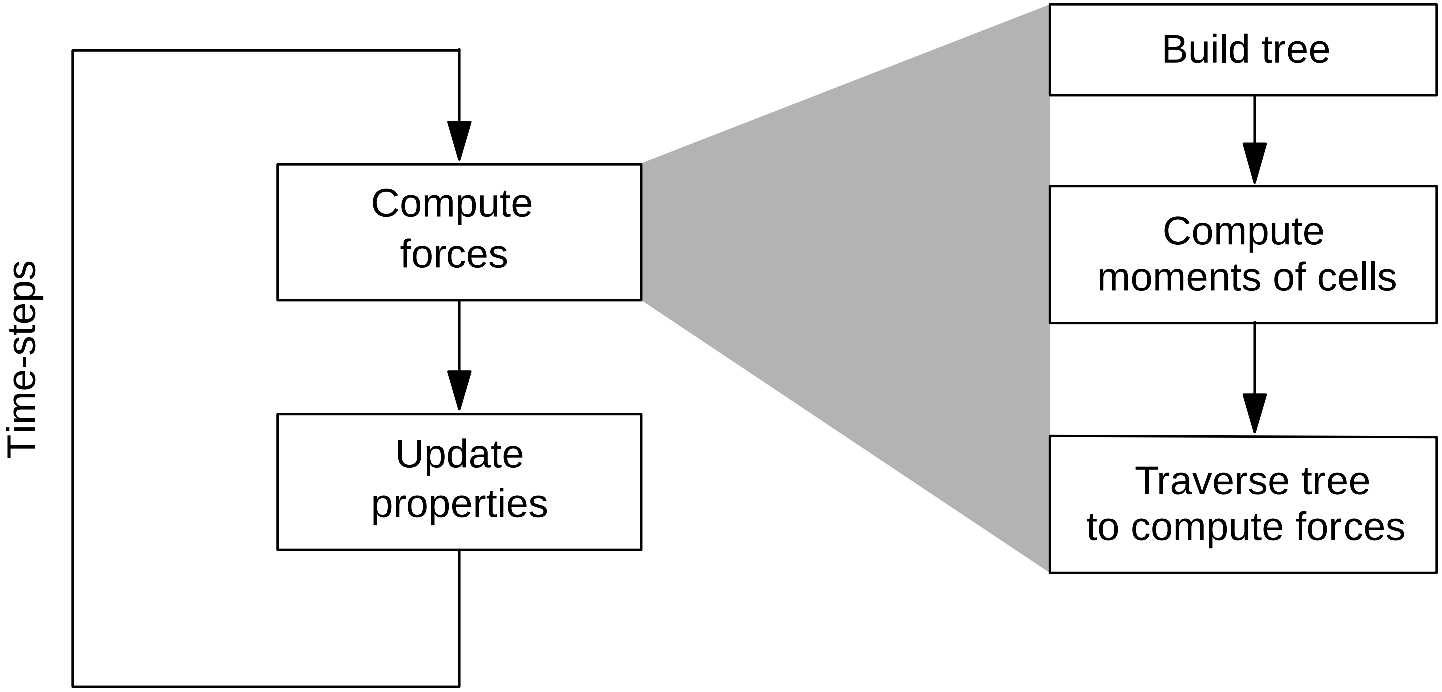
(b) Quadtree representation

Recursive Division of Two-dimensional Space

Locality Goal:

Bodies close together in space should be on same processor

Barnes-Hut Algorithm



- **Main data structures: array of bodies, of cells, and of pointers to them**
  - **Each body/cell has several fields: mass, position, pointers to others**
  - **pointers are assigned to processes**

The Need For Dynamic Tasking: Rendering Scenes by Ray Tracing

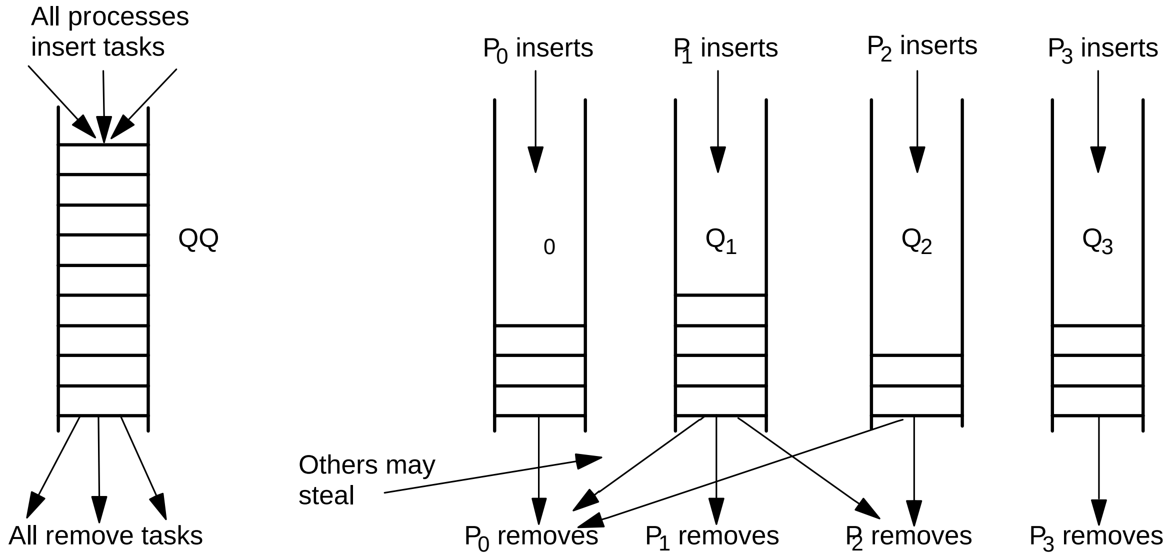
- Shoot rays into a scene through pixels in image plane.
- Follow their paths:
  - They bounce around as they strike objects:
    - They generate new rays:
      - Resulting in a ray tree per input ray and <u>thus more computations</u> (tasks).
- Result is color and opacity for that pixel.
- Parallelism across rays.
  - Parallelism here is <u>unpredictable statically</u>.
  - <u>Dynamic tasking needed for load balancing.</u>

Dynamic Tasking with Task Queues

Centralized versus **distributed** queues.

Task stealing with distributed queues.

- Can compromise communication and data locality (e.g in SAS), and increase synchronization wait time.
- Whom to steal from, how many tasks to steal, ...
- Termination detection (all queues empty).
- Load imbalance possible related to size of task.
  - Many small tasks usually lead to better load balance



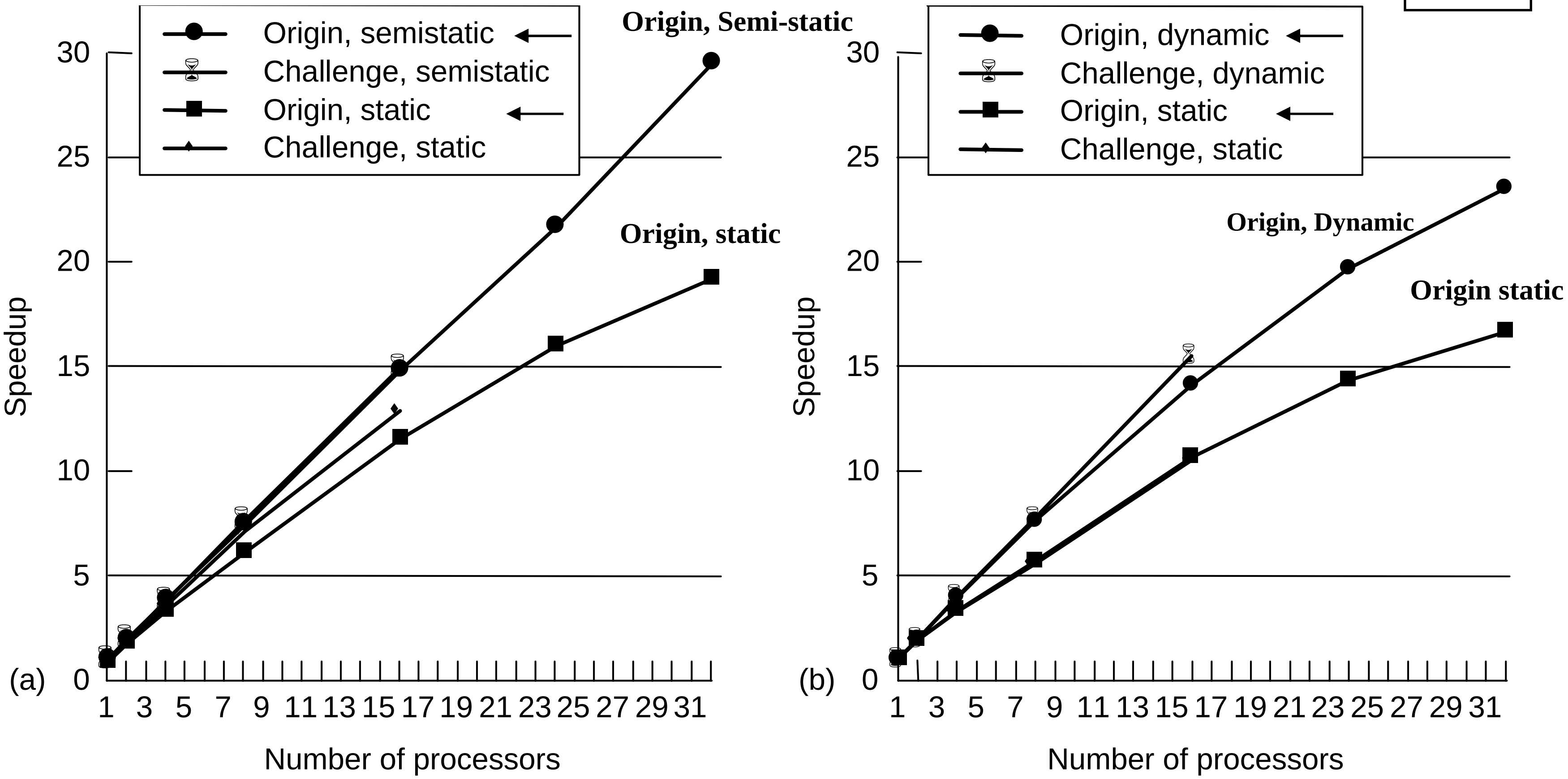
Centralized Task Queue

Distributed Task Queues (one per process

Performance Impact of Dynamic Assignment

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

NUMA



Barnes-Hut 512k particle

Ray tracing

Assignment: Determining Task Granularity

Recall that parallel task granularity:

Amount of work or computation associated with a task.

General rule:

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

But potentially better load balance

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

- **Other overheads are also affected by task size too, particularly with dynamic mapping (tasking) using task queues:**
  - **Small tasks -> More Tasks -> More dynamic mapping overheads.**

A task only executes on one processor to which it has been mapped or allocated

Reducing Serialization/Synch Wait Time

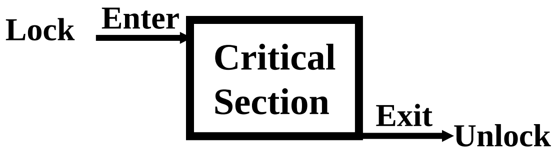
Requires careful assignment and orchestration (and scheduling ?)

Reducing Serialization/Synch wait time in Event synchronization:

i.e Ordering

- Reduce use of conservative synchronization e.g. :
  - Fine point-to-point synchronization instead of barriers (if possible),
  - or reduce granularity of point-to-point synchronization (specific elements instead of entire data structure).
- But fine-grained synch more difficult to program, more synch operations.

Reducing Serialization in Mutual exclusion:



1 – Separate locks for separate data
  - e.g. locking records in a database instead of locking entire 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

2 – Smaller, less frequent critical sections

e.g use of local difference in ocean example

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

3 – Stagger critical sections in time (on different processors).

i.e critical section entry occur at different times

Partitioning for Performance:

Implications of Load Balancing/Synch Time Reduction

Extends speedup limit expression to:

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

Generally load balancing is the responsibility of software

For dynamic tasking

But: **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. multiple-stolen tasks**

=> <u>Hardware shared address space advantageous here</u>

- **Efficient support for point-to-point communication.**
  - **Software layers + hardware (network) support.**

+ CA

Partitioning for Performance:

Reducing Inherent Communication

Inherent Communication : communication between tasks inherent in the problem/parallel algorithm for a given partitioning/assignment (to tasks)

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

Focus here is on reducing interprocess communication inherent in the problem:

i.e inherent communication

- Determined by assignment of parallel computations to tasks/processes.
- Minimize c-to-c ratio while maintaining a good load balance among tasks/processes.
- Actual communication can be greater than inherent communication.

- As much as possible, assign tasks that access same data to same process (and processor later in mapping).

Processor Affinity

Important in SAS NUMA Architectures

- Optimal solution (partition) to reduce communication and achieve an optimal load balance is NP-hard in the general case.
- Simple heuristic partitioning solutions may work well in practice:
  - Due to specific dependency structure of applications.
  - Example: Domain decomposition

Next →

Example Assignment/Partitioning Heuristic:

Domain Decomposition

Domain: Physical domain of problem or input data set

- **Initially used in data parallel scientific computations such as (Ocean) and pixel-based image processing to obtain a good load balance and c-to-c ratio.**

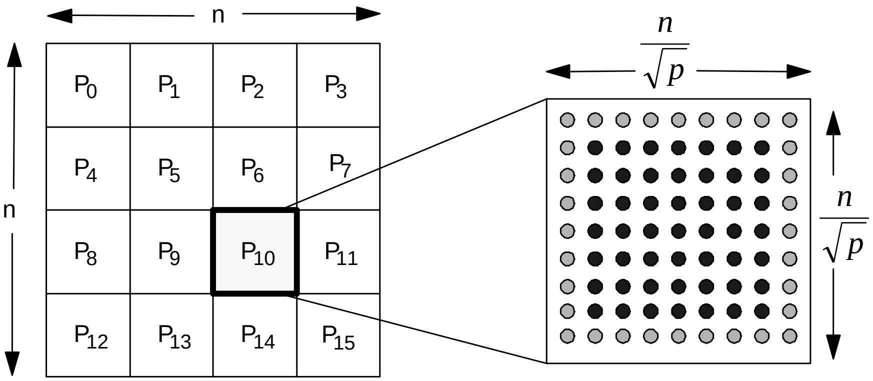
and other usually predictable computations tied to a physical domain/data set

How?

The task assignment is achieved by decomposing the physical domain or data set of the problem.

Such assignment often done statically for predictable computations

- **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 2D grid computation** **(as in ocean example)**



Block Decomposition

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

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

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

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

p = Number of tasks/processes here = p= 4 x 4 = 16

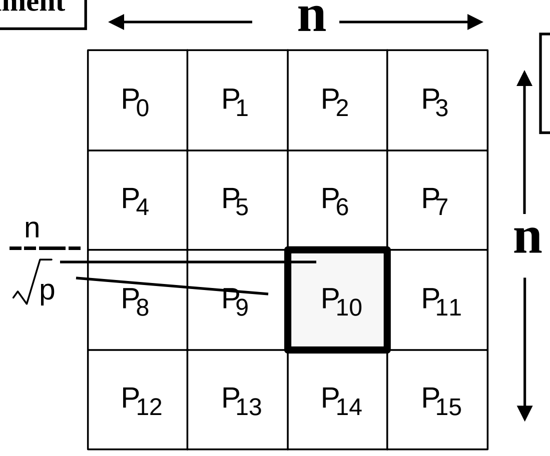
Domain Decomposition (continued)

Best domain decomposition depends on information requirements

Nearest neighbor example: i.e group or "strip" of (contiguous) rows

– **block versus strip domain decomposition:**

Block Assignment

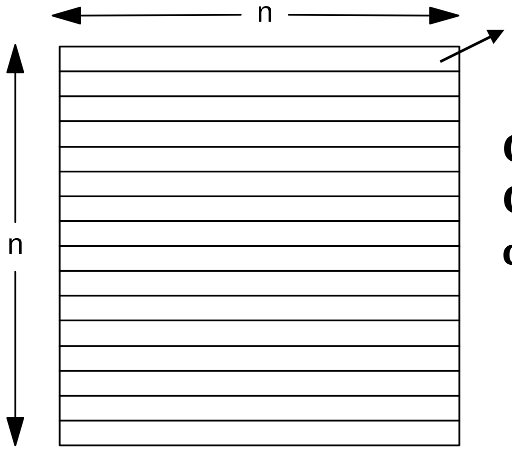


For block assignment

Comm. $4n/\sqrt{p}$

Comp. n^2/p

Strip assignment



n/p rows per task

Communication = 2n
Computation = n^2/p
c-to-c ratio = 2p/n

This "strip assignment" is the assignment used in 2D ocean equation solver example (lecture 4)

Block Decomposition

Strip (Group of rows) Decomposition

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

Which C-to-C ratio is better?

Application dependent: strip may be better in some cases

Often n >> p

Finding a Domain Decomposition

Four possible methods:

More Work

1 • **Static, by inspection:**
  - Computation must be predictable: e.g grid example above, and Ocean

    and low-level (pixel-based) image processing

2 • **Static, but not by inspection:**
  - Input-dependent, require analyzing input structure
    - Before start of computation once input data is known.
  - E.g sparse matrix computations, data mining

Characterized by non-uniform data/computation distribution

3 • **Semi-static (periodic repartitioning):**
  - Characteristics change but slowly; e.g. Barnes-Hut

4 • **Static or semi-static, with dynamic task stealing**
  - Initial decomposition based on domain, but highly unpredictable computation; e.g ray tracing

Implications of Communication

- **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 denominator in c-to-c 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 <u>structure</u> and <u>cost</u> as well:**

<u>Communication Cost</u> = Time added to parallel execution time as a result of communication

From lecture 2

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

→ **<u>Need to keep communication balanced across processors as well</u>.**

c-to-c = communication to computation ratio

Partitioning for Performance: Reducing Extra Work (Overheads)

Must also be balanced among all processors

- **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 (combining) messages, allowing effective naming

More on this a bit later in the lecture

- **Architectural Implications:**
  - Reduce by making communication and orchestration efficient (e.g hardware support of primitives ?)

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

Summary of Parallel Algorithms Analysis

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

    + Synch Wait Time

For PRAM:

$$\text{Speedup}_{\textbf{PRAM}} \leq \frac{\text{Sequential Instructions}}{\underset{\textit{(on any processor)}}{\textit{Max}}\left(\text{Instructions + Synch Wait Time + } \underbrace{\text{Extra Instructions}}_{\text{extra work/computation not in sequential version}}\right)}$$

PRAM Advantages/ Disadvantages

  - **Useful for parallel algorithm development, but possibly unrealistic for real parallel program performance evaluation.**
    - **Ignores communication and also the imbalances it causes**
    - **Can lead to poor choice of partitions as well as orchestration when targeting real parallel systems.**

Limitations of Parallel Algorithm Analysis

i.e communication between tasks inherent in the problem/parallel algorithm for a given partitioning/assignment (to tasks)

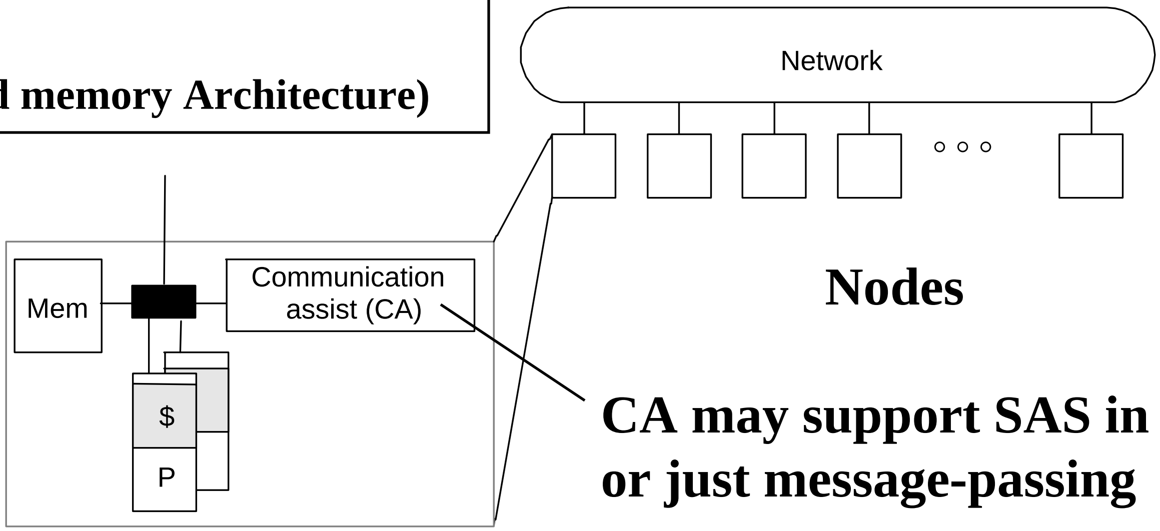
- Inherent communication in a parallel algorithm is not the only communication present:
  - Artifactual "extra" communication caused by program implementation and architectural interactions can even dominate.
  - Thus, actual amount of communication may not be dealt with adequately

    i.e If artifactual communication is not accounted for
- Cost of communication determined not only by amount: +
  - Also how communication is structured and overlapped. +
  - Cost of communication (primitives) in system +
    - Software related and hardware related (network) including CA
- Both are architecture-dependent, and addressed in orchestration step.

Generic Multiprocessor Architecture

If SAS is natively supported by this generic architecture:
⟶ NUMA
(Distributed Shared memory Architecture)

Scalable network.



Nodes

CA may support SAS in hardware or just message-passing

Computing Nodes:

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

- **Network interface and communication controller.**

Scalable Network.

Extended Memory-Hierarchy View of Generic Multiprocessors

SAS support Assumed

Registers → Local Caches → Local Memory $\xrightarrow{\text{Network}}$ Remote Caches
Remote Memory
(Communication)

- **Levels in extended hierarchy:**
  - ^{1} **Registers, ^{2}caches, ^{3} local memory, ^{4}remote memory (over network)**
  - **Glued together by communication architecture**
  - **Levels communicate at a certain granularity of data transfer. (e.g. Cache blocks, pages etc.)**

i.e Minimum size of data transferred between levels

- **Need to exploit spatial and temporal locality in hierarchy**
  - **Otherwise artifactual (extra) communication may also be caused**
  - **Especially important since communication is expensive**

extended

Why?

Over network

This extended hierarchy view is more useful in distributed shared memory (NUMA) parallel architectures

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 + local and remote caches
  - Management of levels:
    - Caches managed by hardware
    - Main memory depends on programming model:
      - SAS: data movement between local and remote transparent
      - Message passing: explicit by sending/receiving messages.
  - Improve performance through architecture or program locality (maximize local data access).

Otherwise artifactual "extra" communication is created

This extended hierarchy view is more useful in distributed shared memory parallel architectures

Artifactual Communication in Extended Hierarchy

Accesses not satisfied in local portion cause communication

- __Inherent Communication__, implicit or explicit, causes transfers:
  - Determined by parallel algorithm/program partitioning

C-to-C Ratio

- __Artifactual "Extra" Communication:__

Why?

Causes of Artifactual "extra" Communication

  - Determined by __program implementation and architecture interactions__
  - __Poor allocation of data__ across distributed memories: data accessed heavily used by one node is located in another node's 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)

i.e zero or no extra communication

- Inherent communication assumes [1] __unlimited capacity__, [2] __small transfers__, [3] __perfect knowledge of what is needed__.

For replication

- More on artifactual communication later; first consider replication-induced further

As defined earlier: __Inherent Communication__ : communication between tasks inherent in the problem/parallel algorithm for a given partitioning/ assignment (to tasks)

Extra Communication and Replication

- **Extra Comm. induced by finite capacity is most fundamental artifact:**

  replication

  - **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).**
- **<u>Classify "misses" in "cache" at any level as for uniprocessors</u>**
  - **1** • ***Compulsory* or *cold* misses (no size effect)**
  - **2** • ***Capacity* misses (yes)**
  - **3** • ***Conflict* or *collision* misses (yes)**
  - **4** • ***Communication* or *coherence* misses (no)**
  - **Each may be helped/hurt by large transfer granularity (spatial locality).**

New C

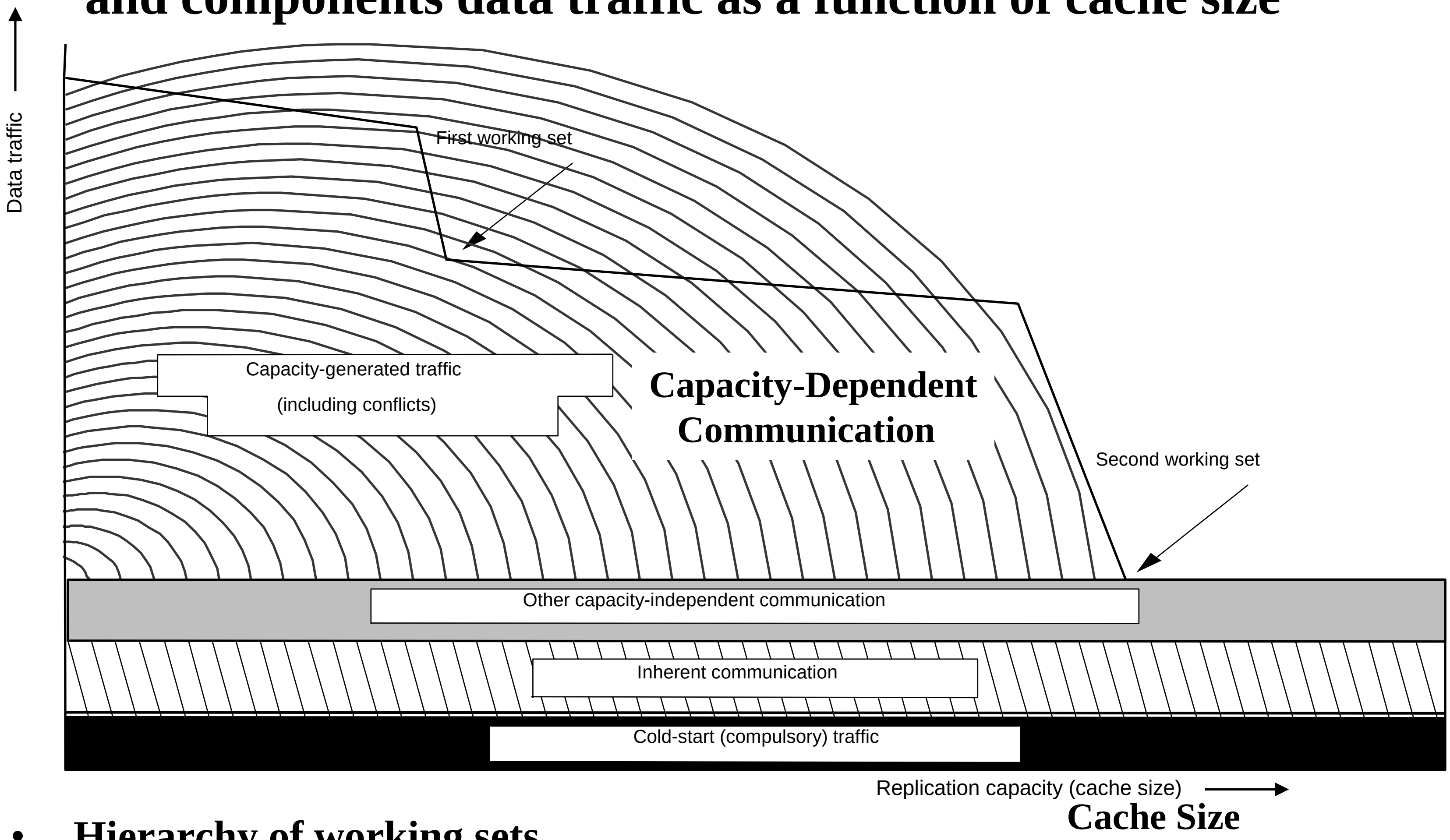
4 Cs

i.e misses that result in extra communication over the network

Distributed shared memory (NUMA) parallel architecture implied here

Working Set Perspective

The data traffic between a cache and the rest of the system and components data traffic as a function of cache size



- **Hierarchy of working sets**
- **Traffic from any type of miss can be local or non-local (communication)**

Distributed shared memory/SAS parallel architecture assumed here

Orchestration for Performance

- **Reducing amount of communication:**
  - **Inherent**: change logical data sharing patterns in algorithm
    - Reduce c-to-c-ratio.

Go back and and change task assignment/partition

  - **Artifactual:** exploit spatial, temporal locality in extended hierarchy
    - Techniques often similar to those on uniprocessors

- **Structuring communication to reduce cost:**

e.g overlap communication with computation or other communication

- **We'll examine techniques for both...**

Reducing Artifactual Communication

- **<u>Message Passing Model:</u>**
  - Communication and replication are both explicit.
  - Even artifactual communication is in explicit messages
    - e.g <u>more data sent in a message than actually needed</u>
- **<u>Shared Address Space (SAS) Model:</u>**
  - More interesting from an architectural perspective
  - Occurs transparently due to interactions of program and system:
    - Caused by sizes of allocation and granularities in extended memory hierarchy (e.g. Cache block size, page size).

i.e. <u>Artifactual Comm.</u>

\+ poor data allocation (NUMA)

- Next, we use shared address space to illustrate issues

→ (distributed memory SAS)

Reducing Artifactual "Extra" Communication

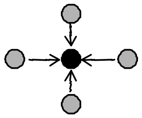
Exploiting Temporal Locality

- <u>**Structure algorithm so working sets map well to hierarchy**</u>
  - **Often techniques to reduce inherent communication do well here**
  - **Schedule tasks for data reuse once assigned**

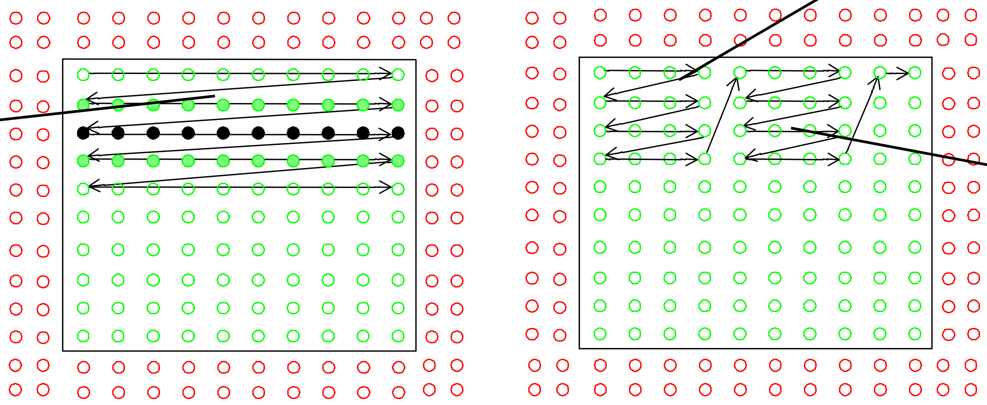
To increase temporal locality

- **Multiple data structures in same phase**
  - **e.g. database records: local versus remote**
- **Solver example: blocking** **(or blocked data access pattern)**

Better Temporal Locality



Unblocked Data Access Pattern



Blocked Data Access Pattern

(a) Unblocked access pattern in a sweep

(b) Blocked access pattern with B = 4

Or more

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

i.e computation with data reuse

Blocked assignment assumed here

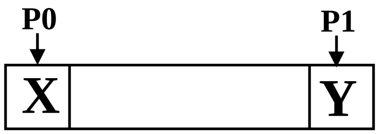
Reducing Artifactual "Extra" Communication

Exploiting Spatial Locality

- Besides capacity, granularities are important:
  - Granularity of allocation *(e.g. page size)*
  - Granularity of communication or data transfer
  - Granularity of coherence *(e.g. cache block size)*

Larger "granularity" when farther from processor

- Major spatial-related causes of artifactual communication:
  - Conflict misses
  - Data distribution/layout (allocation granularity)
  - Fragmentation (communication granularity)
  - False sharing of data (coherence granularity)



Cache block falsely shared between P0 and P1

- All depend on how spatial access patterns interact with data structures/architecture:
  - Fix problems by modifying data structures, or layout/alignment (as shown in example next)

Fix?

- Examine later in context of architectures
  - One simple example here: data distribution in SAS solver

Next

Distributed memory (NUMA) SAS assumed here

Reducing Artifactual “Extra” Communication

Spatial Locality Example

- Repeated sweeps over elements of 2D grid, block assignment, Shared address space; (SAS)
- In Distributed memory: A memory page is allocated in one nodes memory
- Natural 2D versus higher-dimensional (4D here) array representation

i.e granularity of data allocation

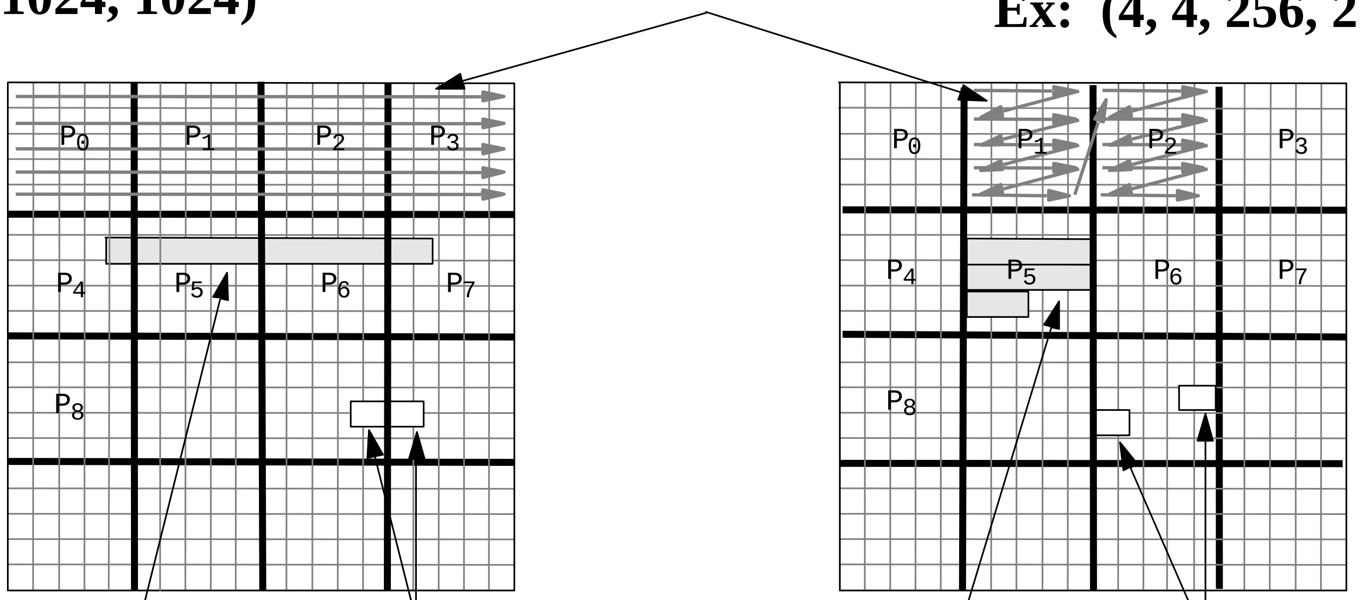
Ex: (1024, 1024)

Ex: (4, 4, 256, 256)

Contiguity in memory layout

2D Array Representation

4D Array Representation



Page straddles partition boundaries: difficult to distribute memory well

Cache block straddles partition boundary

Page does not straddle partition boundary

Cache block is within a partition

Two-Dimensional (2D) Array

Four-Dimensional (4D) Array

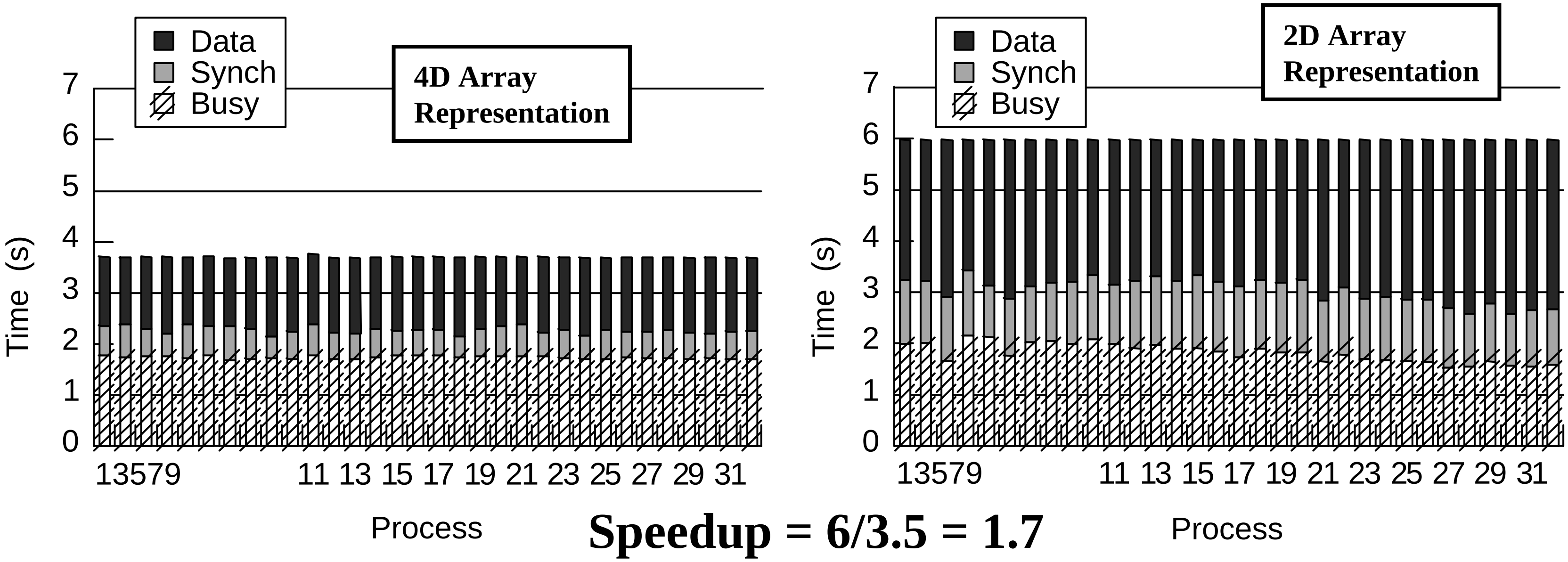
(Generates more artifactual “extra”communication)

SAS assumed here

Performance Comparison Next →

Execution Time Breakdown for Ocean on a 32-processor Origin2000

1026 x 1026 grids with block partitioning on 32-processor Origin2000



Speedup = 6/3.5 = 1.7

Four-dimensional (4D) arrays **Two-dimensional (2D) arrays**

- **4D grids much better than 2D, despite very large caches on machine (4MB L2 cache)**
  - **data distribution is much more crucial on machines with smaller caches**

Thus less replication capacity

- **Major bottleneck in this configuration is time waiting at barriers**
  - **imbalance in memory stall times as well**

Tradeoffs with Inherent Communication

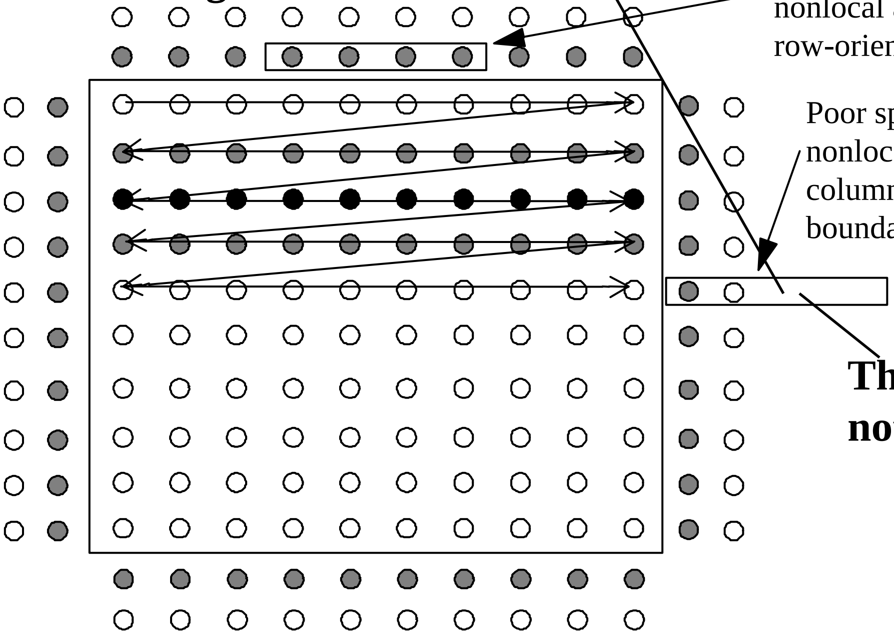
i.e block assignment

Partitioning grid solver: blocks versus rows (i.e strip assignment)

- **Blocks still have <u>a spatial locality problem</u> on remote data**
- **Row-wise (strip) can perform better despite worse inherent c-to-c ratio**

As shown next

Block Assignment:



Good spatial locality on nonlocal accesses at row-oriented boundary

Poor spatial locality on nonlocal accesses at column-oriented boundary

These elements not needed

- Result depends on *n* and *p*

<u>Results to show this next</u>

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

Example Performance Impact

Equation solver on SGI Origin2000 (distributed shared memory)

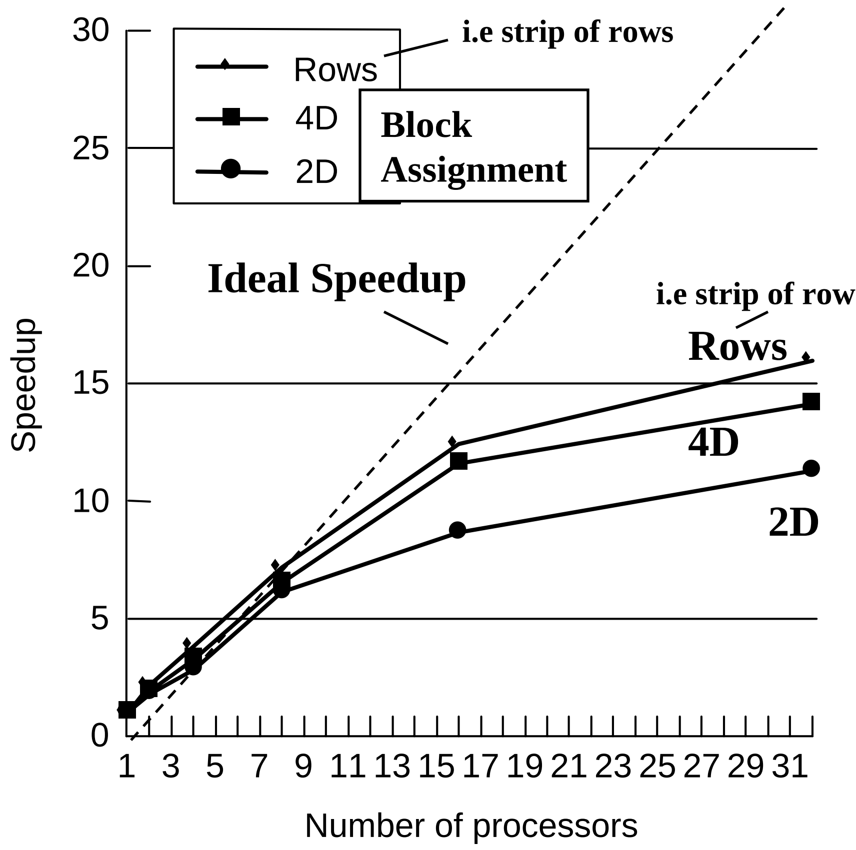
rr = Round Robin Page Distribution

Rows = Strip Assignment

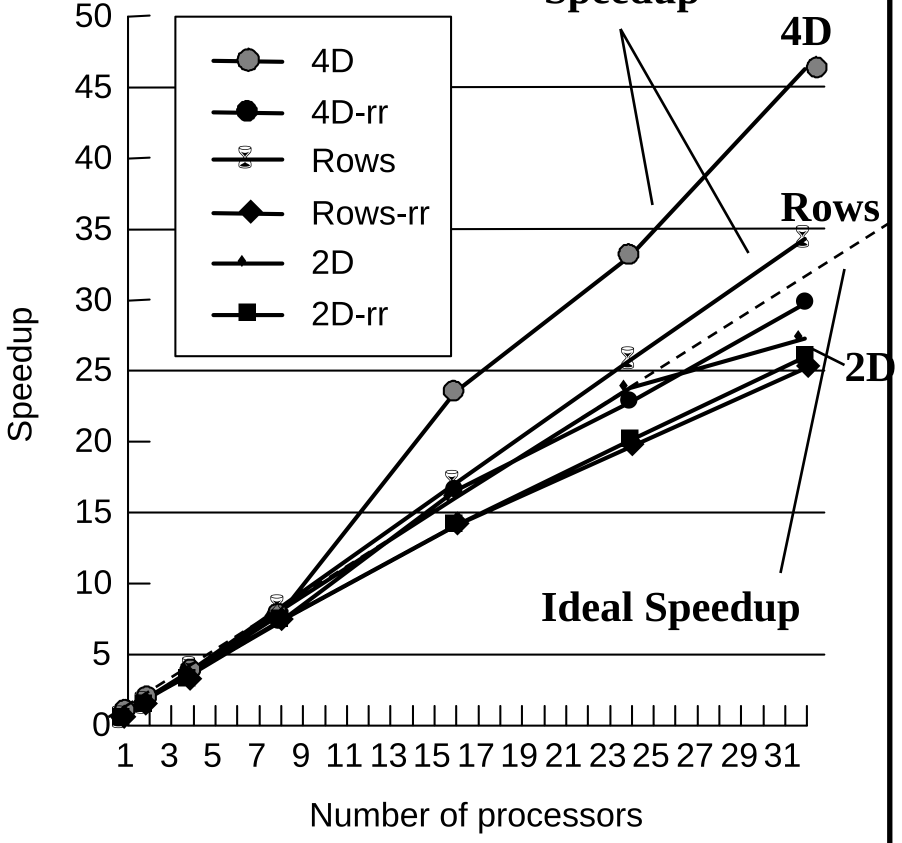
2D, 4D block assignment

Why?

Super-linear Speedup



514 x 514 grids



12k x 12k grids

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

Structuring Communication

Given amount of comm. (inherent or artifactual), goal is to reduce cost ↓

- **Total cost of communication as seen by process:**

Want to reduce ↓

Want to reduce

$$C = f * (\overbrace{\underbrace{o + l + \frac{n_c/m}{B} + t_c}_{\text{Latency of a message}} - overlap}^{\text{Cost of a message}})$$

Reduce ?

Want to increase ↑

Want to reduce ↓

- **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** **One may consider m = f**
- **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 other comm.**

n_c/m average length of message

- **Portion in parentheses is <u>cost of a message</u> (as seen by processor)**
- **That portion, ignoring overlap, is <u>*latency* of a message</u>**
- **<u>Goal:</u> <u>1- reduce terms in communication latency and</u>**
  <u>2- increase overlap</u>

<u>Communication Cost:</u> Actual time added to parallel execution time as a result of communication

Reducing Cost of Communication:

f * o

Reducing Overall Communication Overhead

i.e total

- Can reduce number of messages *f* or reduce overhead per message *o*
- Message overhead, *o* is usually determined by hardware and system software (implementation cost of comm. primitives)
  - Program should try to reduce number of messages m by <u>combining messages.</u> (fewer messages)
  - More control when communication is explicit (message-passing).

Reduce total comm. overhead, How?

- <u>Combining data into larger messages:</u> *to reduce number of messages, f*
  - Easy for regular, coarse-grained communication
  - Can be difficult for irregular, naturally fine-grained communication.
    - <u>May require changes to algorithm and extra work</u> e.g duplicate computations
      - Combining data and determining what and to whom to send
    - <u>May increase synchronization wait.</u>

Longer synch wait to get more results data computed to send in larger message

Reducing Cost of Communication:

Reducing Network Delay

- Total network delay component = $f * l = f*h*t_h$
  - h = number of hops traversed in network
  - t_h = link+switch latency per hop

in route from source to destination

Depends on
Mapping
Network Topology
Network Properties

- Reducing f: Communicate less, or make messages larger

Thus fewer messages

- Reducing h *(number of hops)*:
  - Map <u>task communication patterns</u> to <u>network topology</u>
    e.g. nearest-neighbor on mesh and ring etc.
  - How important is this?
    - <u>Used to be a major focus of parallel algorithm design</u>
    - Depends on number of processors, how t_h, compares with other components, network topology and properties
    - <u>Less important on modern machines</u>
      - (e.g. Generic Parallel Machine)

Graph Matching Problem

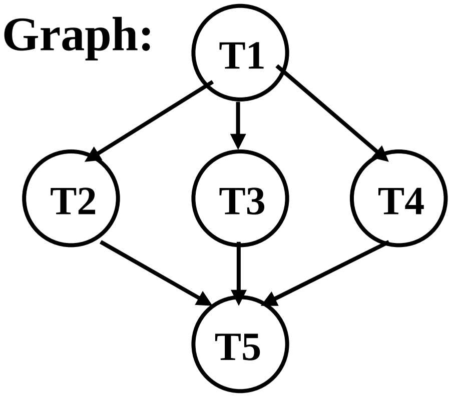
Optimal solution is NP problem

Where equal communication time/delay between any two nodes is assumed (i.e symmetric network)

Mapping of Task Communication Patterns to Topology Example

Reducing Network Delay: Reduce Number of Hops

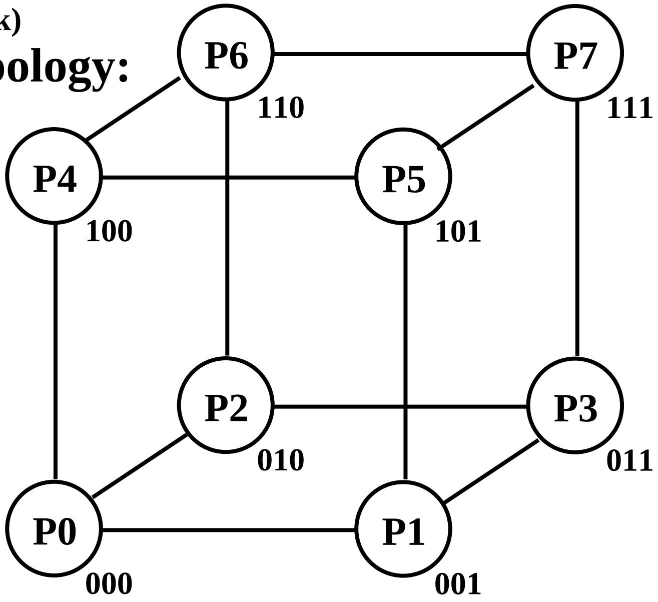
Task Graph:



(network)

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 Cost of Communication:

Reducing Contention t_c

- **All resources have nonzero occupancy (busy time):**
  - **Memory, communication assist (CA), network link, etc.**
    - **Can only handle so many transactions per unit time.**
  - **Contention results in queuing delays at the busy resource.**

i.e Occupancy **i.e contended**

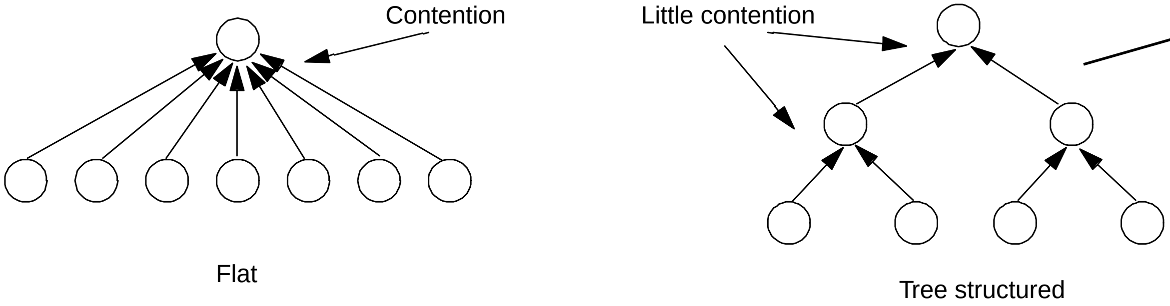
- **Effects of contention:**
  - **Increased end-to-end cost for messages.** **e.g delay, latency**
  - **Reduced available bandwidth for individual messages.**
  - **Causes imbalances across processors.**
- **Particularly insidious performance problem:**
  - **Easy to ignore when programming**
  - **Slows down messages that don't even need that resource**
    - **By causing other dependent resources to also congest** **Ripple effect**
  - **Effect can be devastating: *Don't flood a resource!***

How?

Types of Contention

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

i.e one point of contention



More on this next lecture - Implementations of barriers

i.e several points of contention

- Module: all-to-all personalized comm. in matrix transpose
  - Solution: stagger access by different processors to same node temporally

How to reduce contention?

- In general, reduce burstiness (smaller messages); may conflict with making messages larger (to reduce number of messages)

Reducing Cost of Communication:

Overlapping Communication

- Cannot afford to stall/wait for high latencies
- Overlap with computation or other communication to hide latency → To reduce communication cost
- Common Techniques:

1 – Prefetching (start access or communication before needed
2 – Block data transfer (may introduce extra communication)
3 – Proceeding past communication (e.g. non-blocking receive)
4 – Multithreading (switch to another ready thread or task)

i.e other ready task of the same problem

- <u>In general these above techniques require:</u>

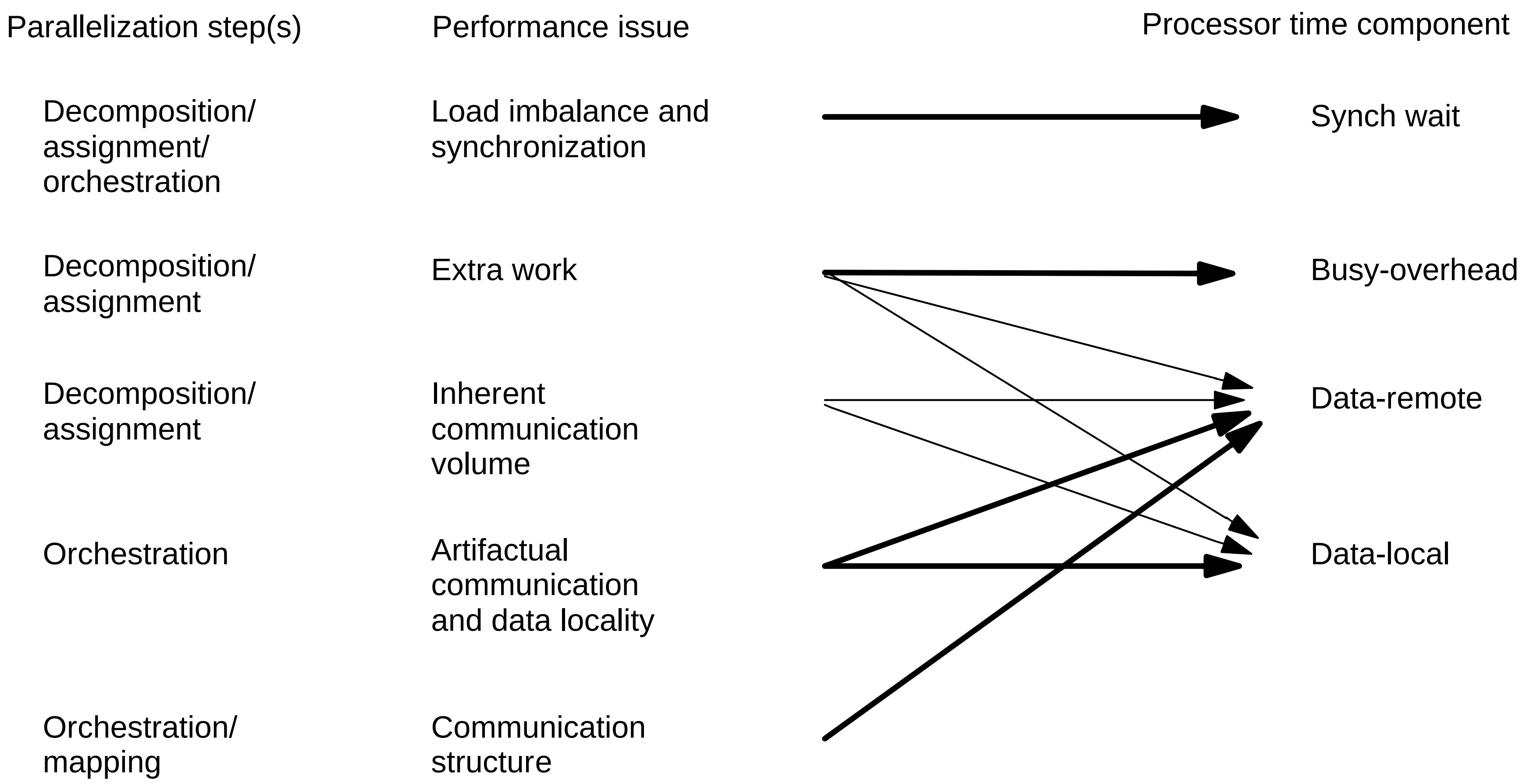
1 – <u>Extra concurrency per node</u> (*slackness*) to find some other computation.
2 – <u>Higher available network bandwidth</u> (for prefetching).
3 – <u>Availability of communication primitives that support overlap.</u>

More on these techniques in PCA Chapter 11

Summary of Tradeoffs

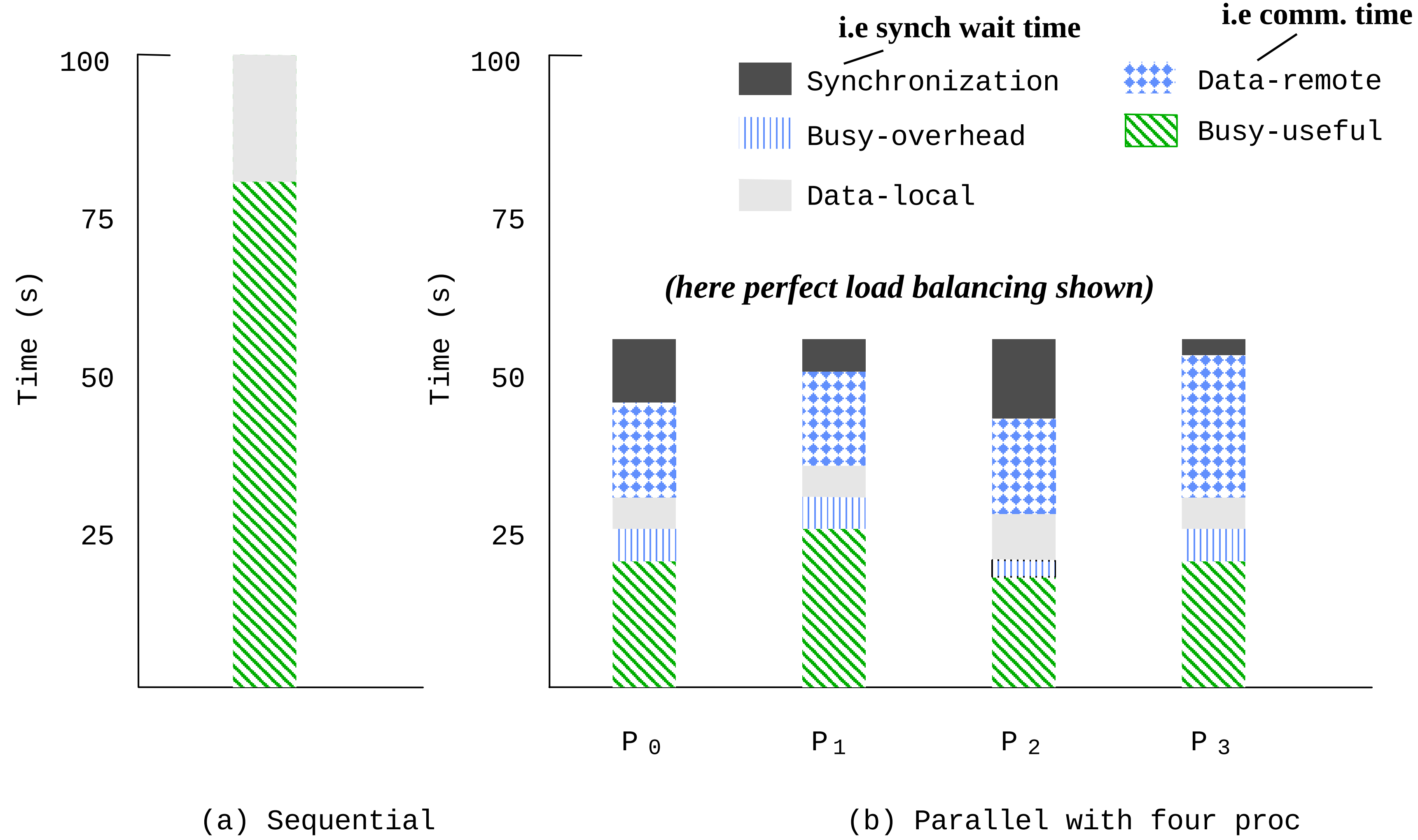
- **Different goals often have conflicting demands:**
  - **Better Load Balance Implies:**
    - **Fine-grain tasks**
    - **Random or dynamic assignment**
  - **Lower Amount of Communication Implies:**
    - **Usually coarse grain tasks**
    - **Decompose to obtain locality: not random/dynamic**
  - **Lower Extra Work Implies:**
    - **Coarse grain tasks**
    - **Simple assignment**
  - **Lower Communication Cost Implies:**
    - **Big transfers: to amortize overhead and latency**
    - **Small transfers: to reduce contention**

Relationship Between Perspectives



Parallelization step(s)	Performance issue	Processor time component
Decomposition/ assignment/ orchestration	Load imbalance and synchronization	Synch wait
Decomposition/ assignment	Extra work	Busy-overhead
Decomposition/ assignment	Inherent communication volume	Data-remote
Orchestration	Artifactual communication and data locality	Data-local
Orchestration/ mapping	Communication structure	

Components of Execution Time From Processor Perspective



(a) Sequential

(b) Parallel with four proc

Summary

$$\textit{Speedup}_{prob}(p) = \frac{Busy(1) + Data(1)}{\underset{\text{(on any processor)}}{\textbf{Max}}\left(Busy_{useful}(p)+Data_{local}(p)+Synch(p)+Date_{remote}(p)+Busy_{overhead}(p) \right)}$$

- Goal is to reduce denominator components
- Both programmer and system have a role to play
- Architecture cannot do much about load imbalance or too much communication
- But it can help:
  - 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

May introduce it , though