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

- Why should architects care?
  - understanding the workloads for their machines
  - hardware/software tradeoffs: where should/shouldn’t architecture help
- Point out some architectural implications
- Architectural techniques covered in rest of class

Programming as Successive Refinement

Not all issues dealt with up front

Partitioning often independent of architecture, and done first

- View machine as a collection of communicating processors
  - balancing the workload
  - reducing the amount of inherent communication
  - reducing extra work
- Tug-o-war even among these three issues

Then interactions with architecture

- View machine as extended memory hierarchy
  - extra communication due to architectural interactions
  - cost of communication depends on how it is structured
- May inspire changes in partitioning

Discussion of issues is one at a time, but identifies tradeoffs

- Use examples, and measurements on SGI Origin2000

Outline

Partitioning for performance

Relationship of communication, data locality and architecture

Programming for performance

For each issue:

- Techniques to address it, and tradeoffs with previous issues
- Illustration using case studies
- Application to grid solver
- Some architectural implications

Components of execution time as seen by processor

- What workload looks like to architecture, and relate to software issues

Applying techniques to case-studies to get high-performance versions

Implications for programming models

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

- 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

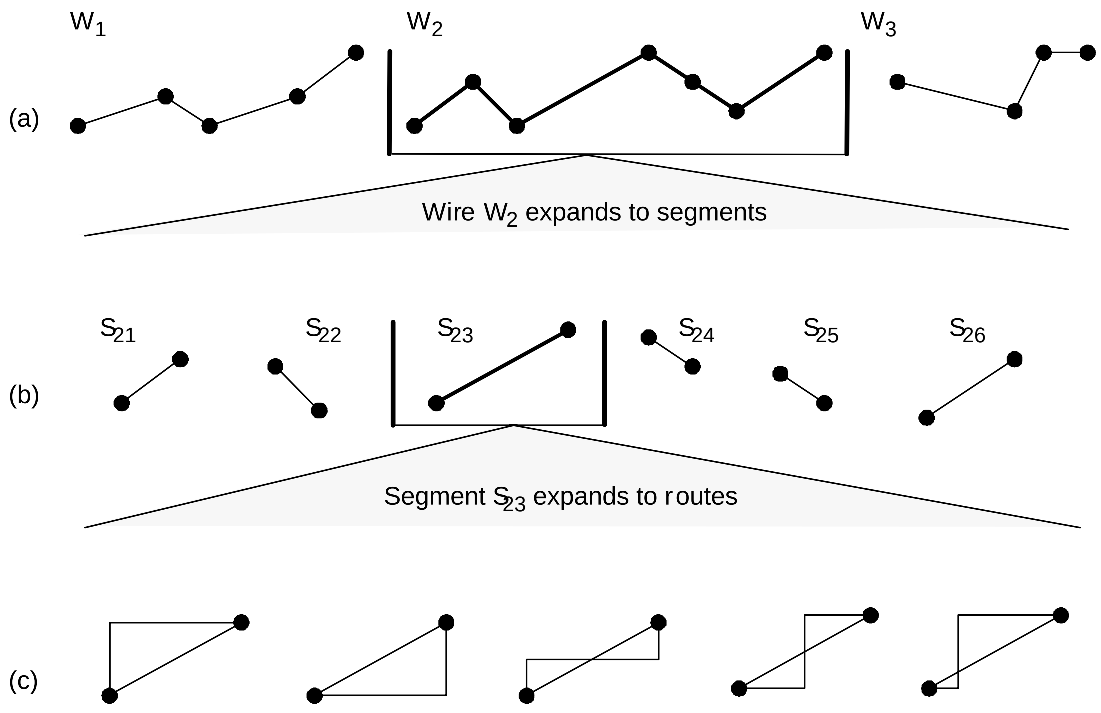
Identifying Concurrency

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



Identifying Concurrency (contd.)

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
- 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

Most scalable programs data parallel (per this loose definition)

- function parallelism reduces synch between data parallel phases

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:

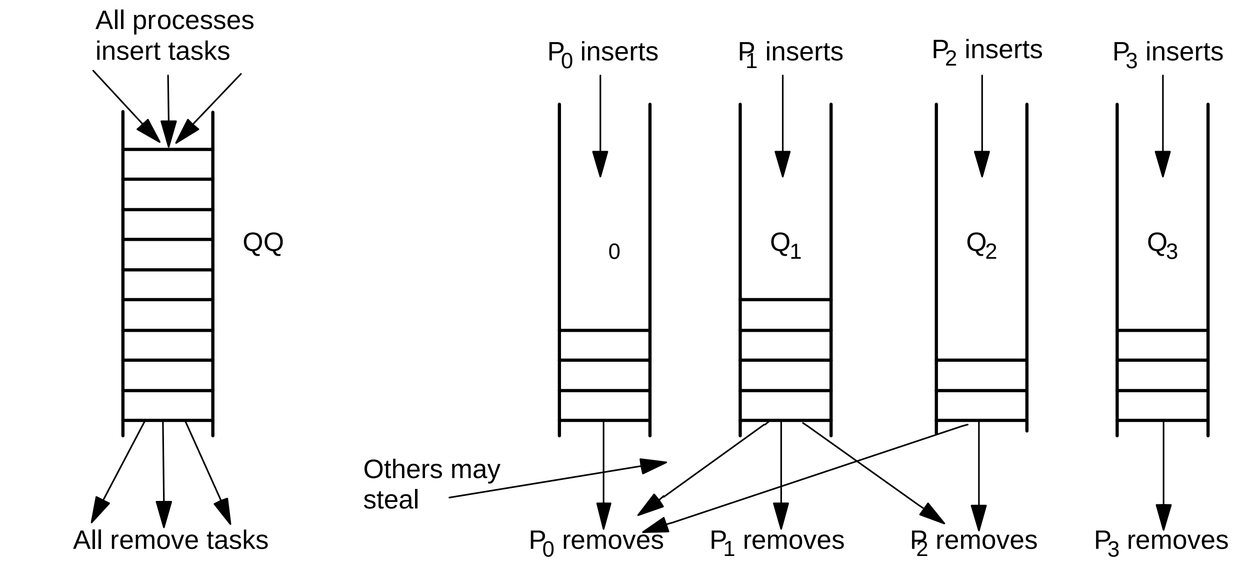
- Deal with unpredictability in program or environment (e.g. Raytrace)
  - computation, communication, and memory system interactions
  - multiprogramming and heterogeneity
  - used by runtime systems and OS too
- Pool of tasks; take and add tasks until done
- E.g. “self-scheduling” of loop iterations (shared loop counter)

Dynamic Tasking with Task Queues

Centralized versus distributed queues

Task stealing with distributed queues

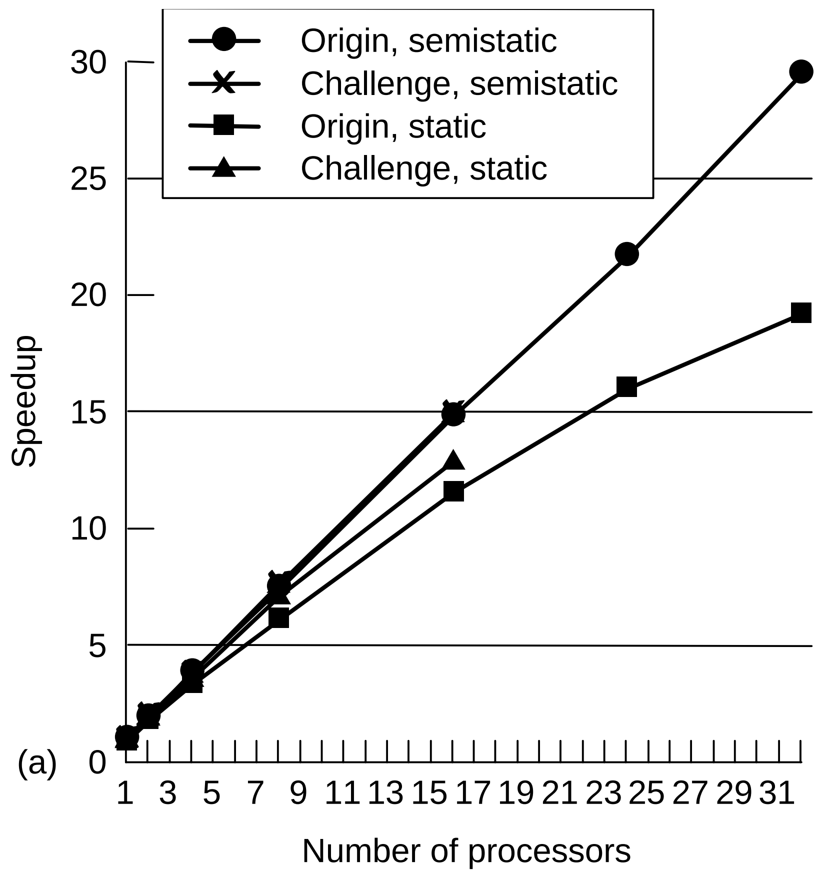
- Can compromise comm and locality, and increase synchronization
- Whom to steal from, how many tasks to steal, ...
- Termination detection
- Maximum imbalance related to size of task

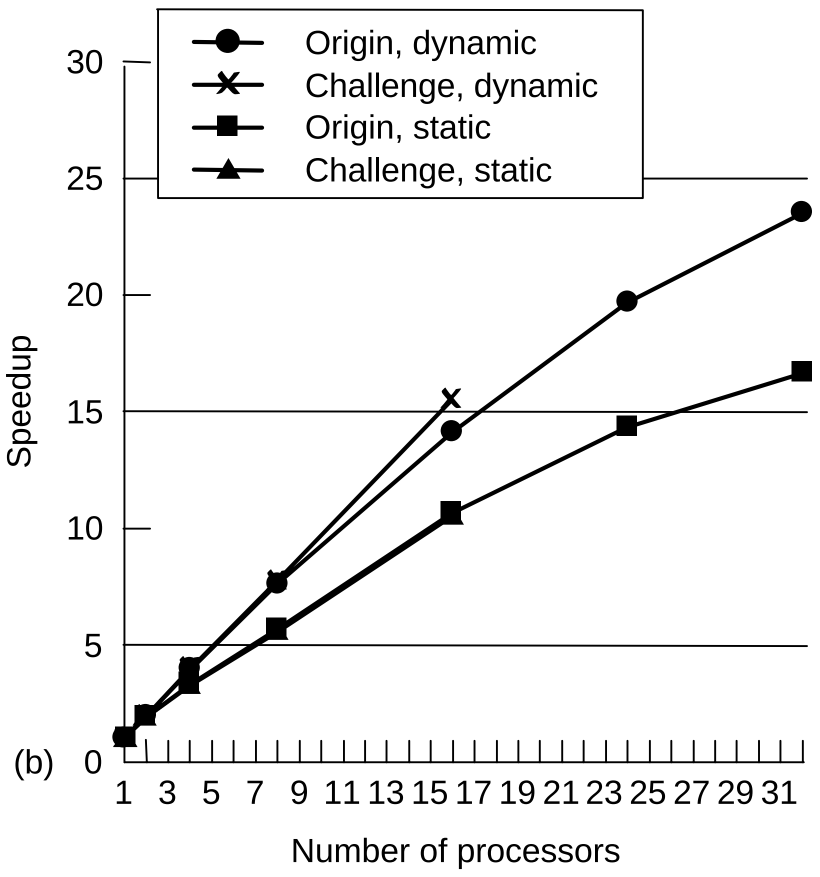


(a) Centralized task queue (b) Distributed task queues (one per process)

Impact of Dynamic Assignment

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





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 comm., contention

Comm., contention actually affected by assignment, not size

- Overhead by size itself too, particularly with task queues

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
  - don’t 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

Implications of Load Balance

Extends speedup limit expression to: $\leq \dfrac{\text{Sequential Work}}{\textit{Max}\ (\text{Work} + \text{Synch Wait Time})}$

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
  - 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

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!

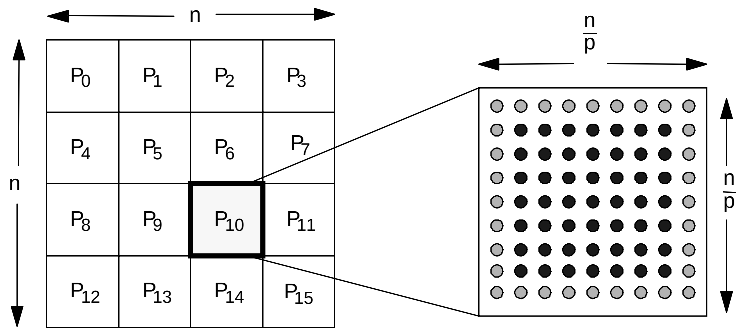
Domain Decomposition

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

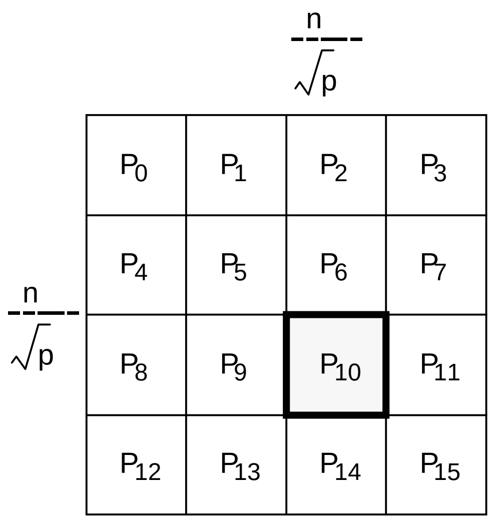


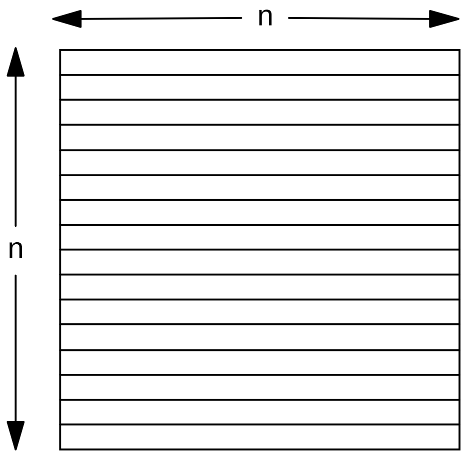
Perimeter to Area comm-to-comp ratio (area to volume in 3-d)

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

Domain Decomposition (contd)

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 (assigning itemsets)

Semi-static (periodic repartitioning)

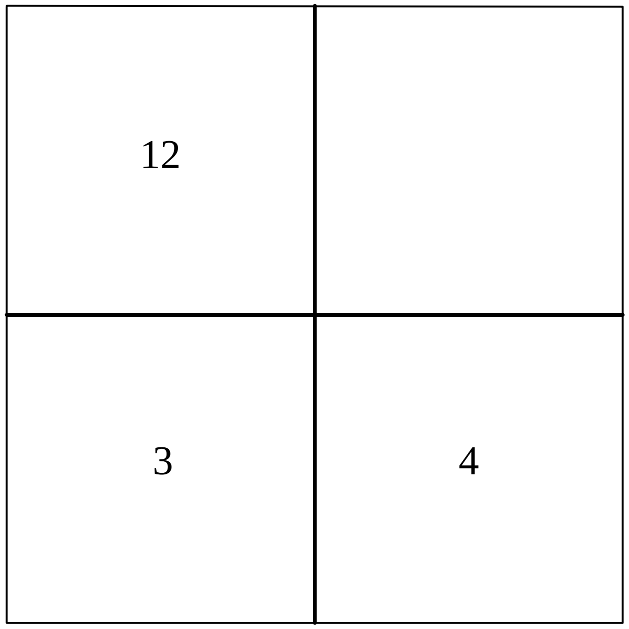
- Characteristics change but slowly; e.g. Barnes-Hut

Static or semi-static, with dynamic task stealing

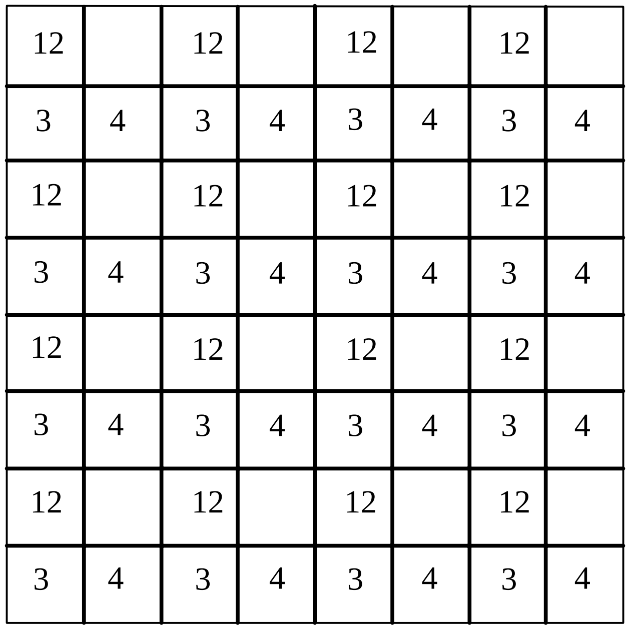
- Initial decomposition, but highly unpredictable; e.g ray tracing

Other Techniques

Scatter Decomposition, e.g. initial partition in Raytrace



Domain decomposition



Scatter decomposition

Preserve locality in task stealing

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

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 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

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

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
- More recent models incorporate comm. costs; BSP, LogP, ...

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
- Prog. model and comm. abstr. affect specific performance tradeoffs

Most of remaining perf. 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

Uniprocessor

Performance depends heavily on memory hierarchy

Time spent by a program

$Time_{prog}(1) = Busy(1) + Data\ Access(1)$

- Divide by cycles to get CPI equation

Data access time can be reduced by:

- Optimizing machine: bigger caches, lower latency...
- Optimizing program: temporal and spatial locality

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 Comm. 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
  - 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.
- More on artifactual later; first consider replication-induced further

Communication and Replication

Comm induced by finite capacity is most fundamental artifact

- Like 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)

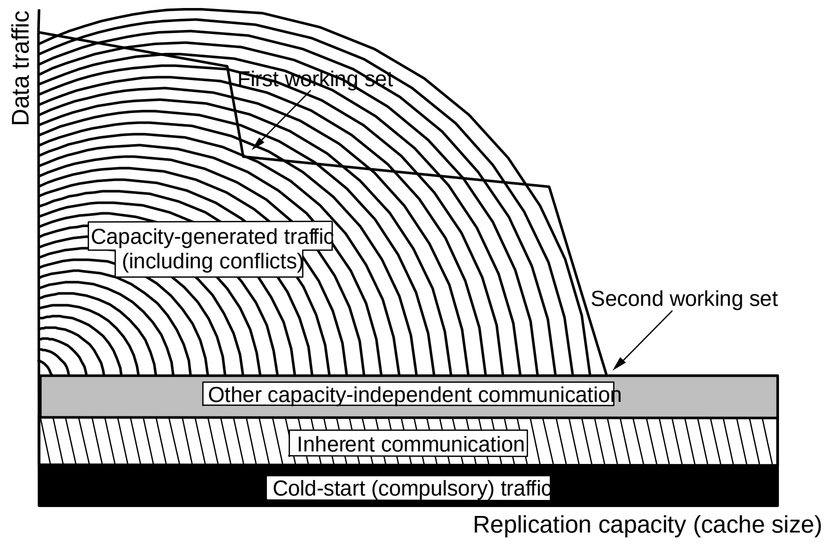
Classify “misses” in “cache” at any level as for uniprocessors

- *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)


Data traffic
First working set
Capacity-generated traffic (including conflicts)
Second working set
Other capacity-independent communication
Inherent communication
Cold-start (compulsory) traffic
Replication capacity (cache size)

- 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

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

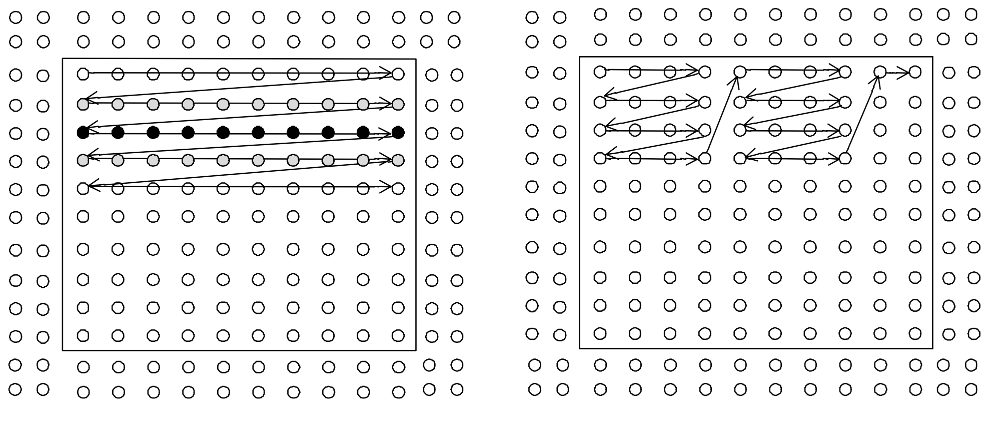
Shared address space model

- 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 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

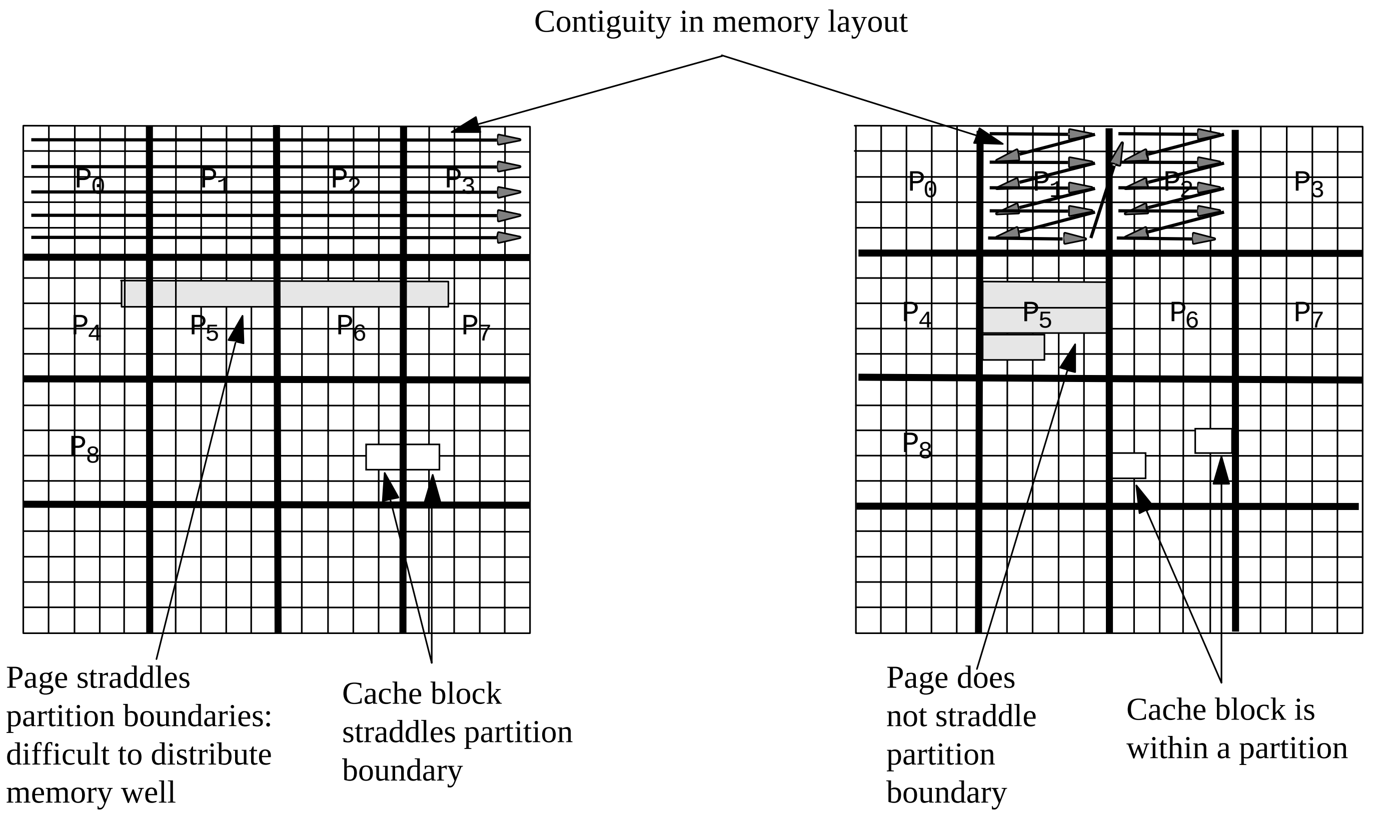
- Fix problems by modifying data structures, or layout/alignment

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
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8
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

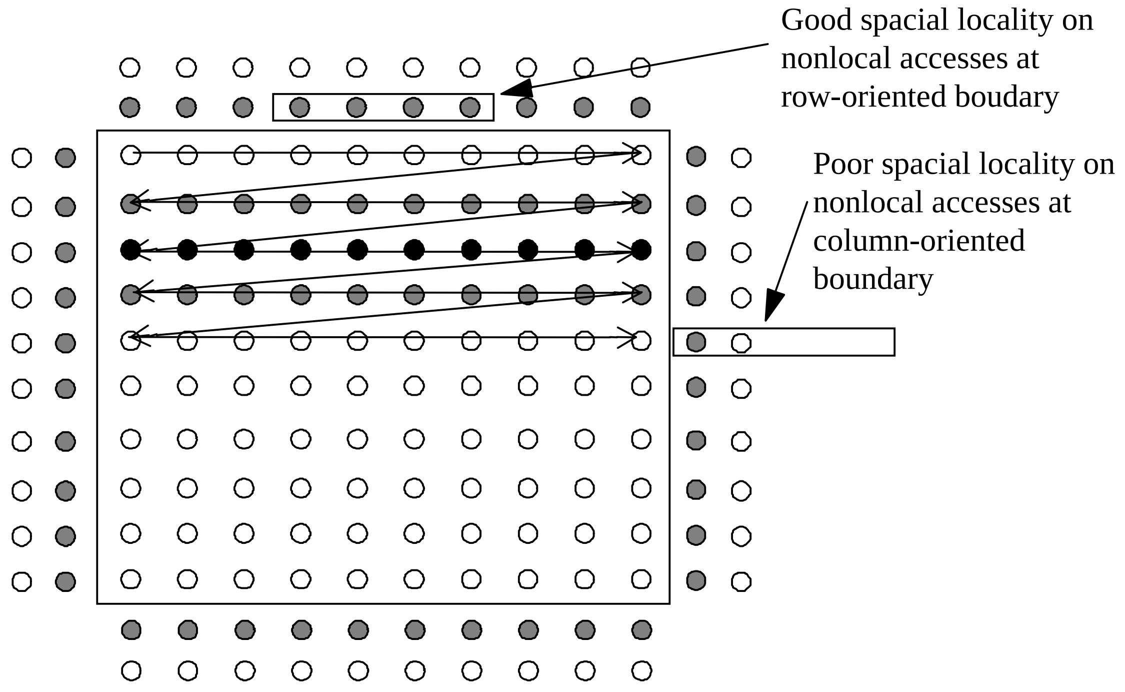
(a) Two-dimensional array

(b) Four-dimensional array

Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows

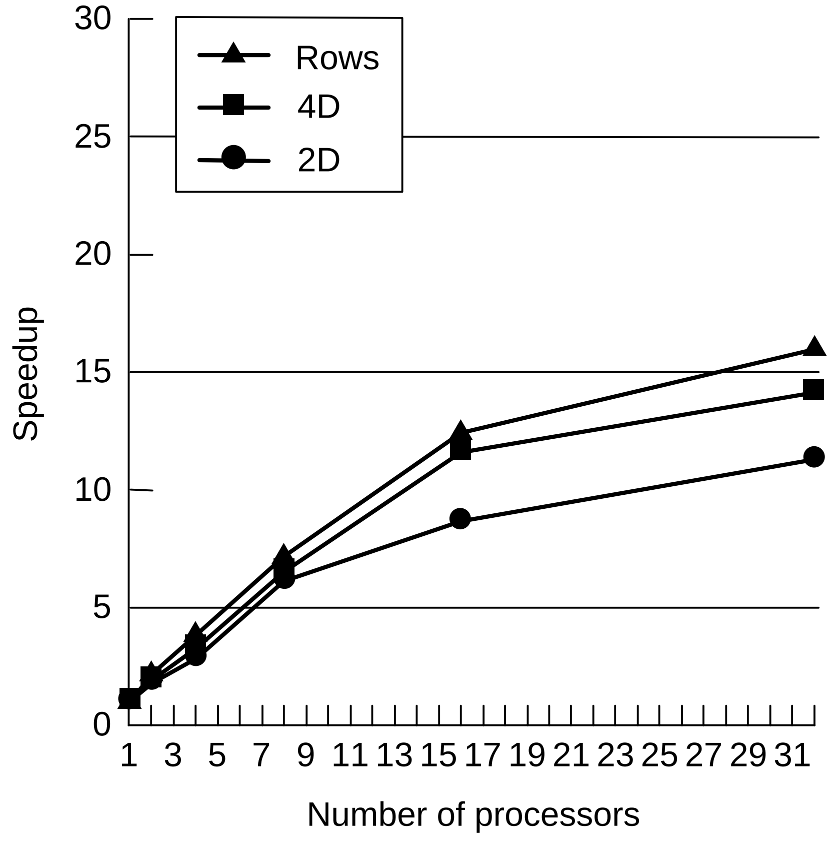
- Blocks still have a spatial locality problem on remote data
- Rowwise can perform better despite worse inherent c-to-c ratio

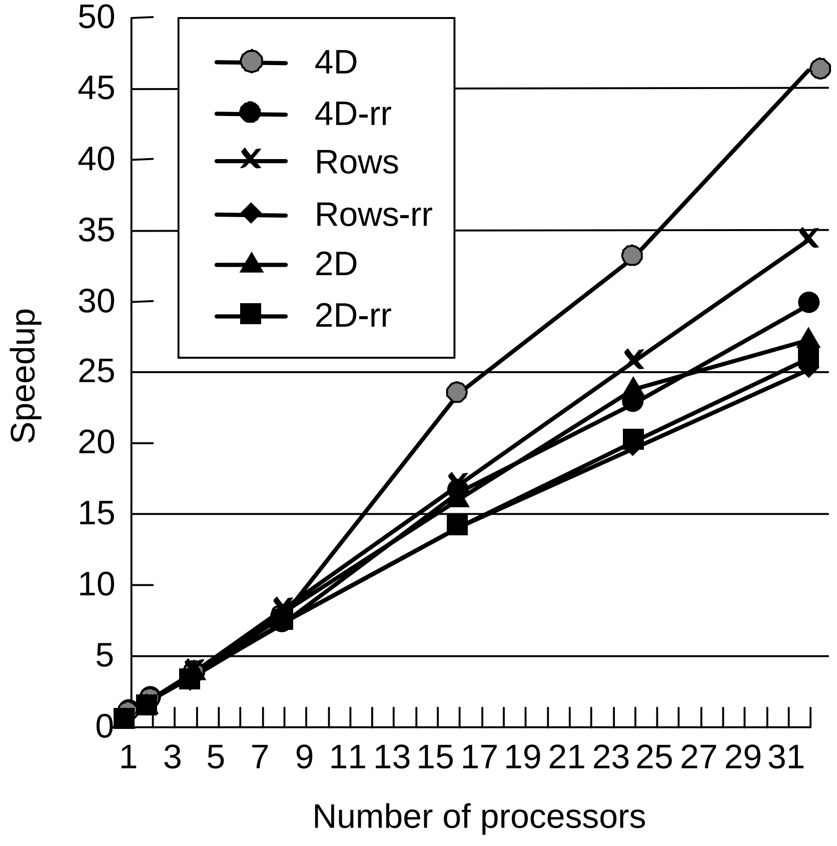

Good spacial locality on nonlocal accesses at row-oriented boudary
Poor spacial locality on nonlocal accesses at column-oriented boundary

- Result depends on *n* and *p*

Example Performance Impact

Equation solver on SGI Origin2000





Architectural Implications of Locality

Communication abstraction that makes exploiting it easy

For cache-coherent SAS, e.g.:

- 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...)

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

<u>Reducing Overhead</u>

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 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
  - will discuss more in implications for programming models later

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 no. of processors, how t_h, compares with other components
  - less important on modern machines
    - overheads, processor count, multiprogramming

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

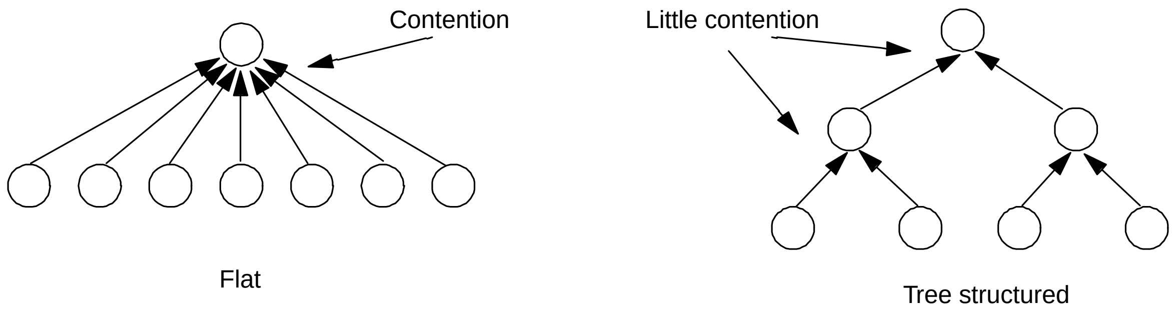
- Easy to ignore when programming
- Slow down messages that don’t even need that resource
  - by causing other dependent resources to also congest
- Effect can be devastating: *Don’t flood a resource!*

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



- 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

- even on uniprocessors!

Overlap with computation or communication to hide latency

Requires extra concurrency (*slackness*), higher bandwidth

Techniques:

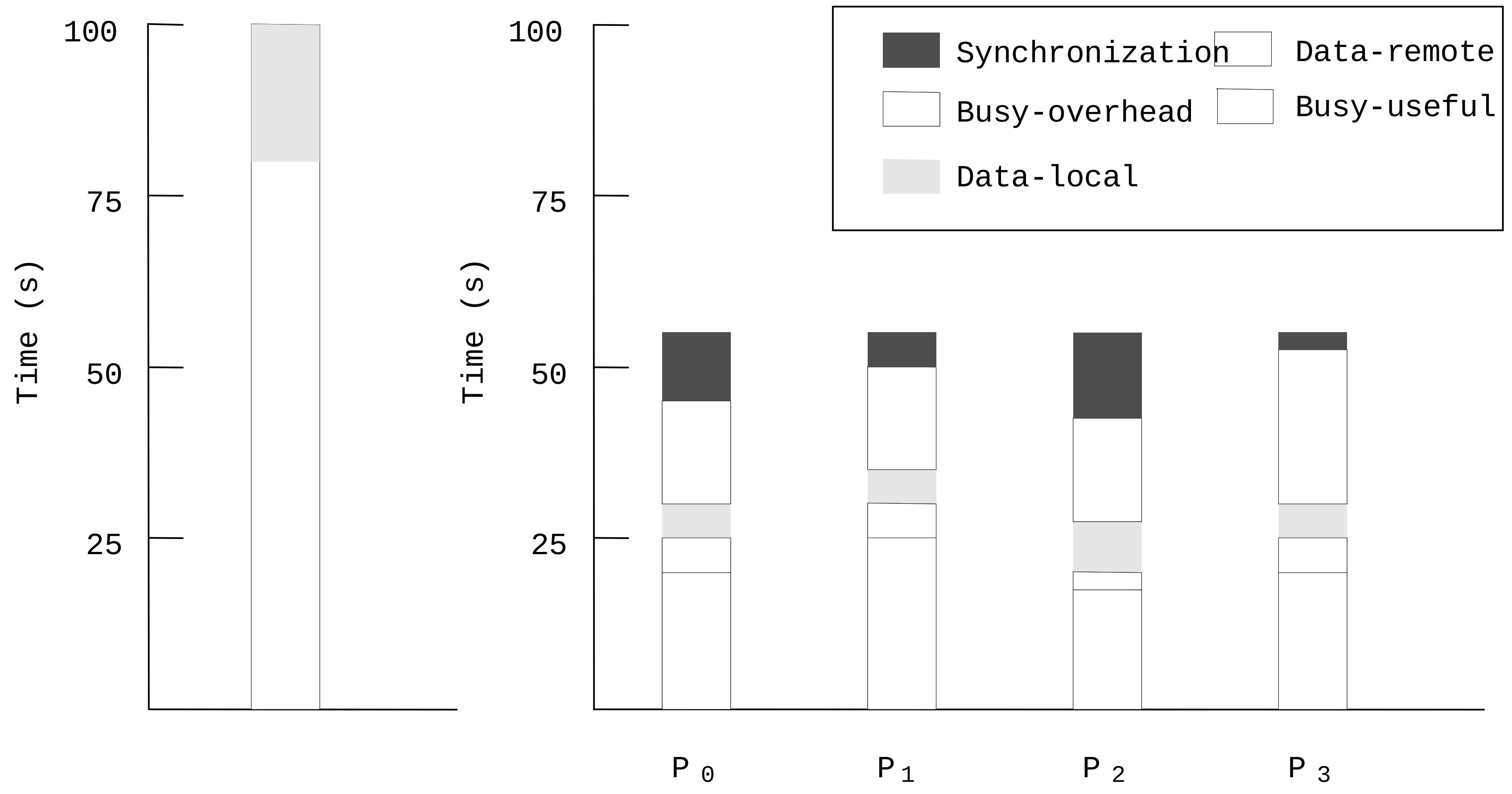
- Prefetching
- Block data transfer
- Proceeding past communication
- Multithreading

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

Processor-Centric Perspective

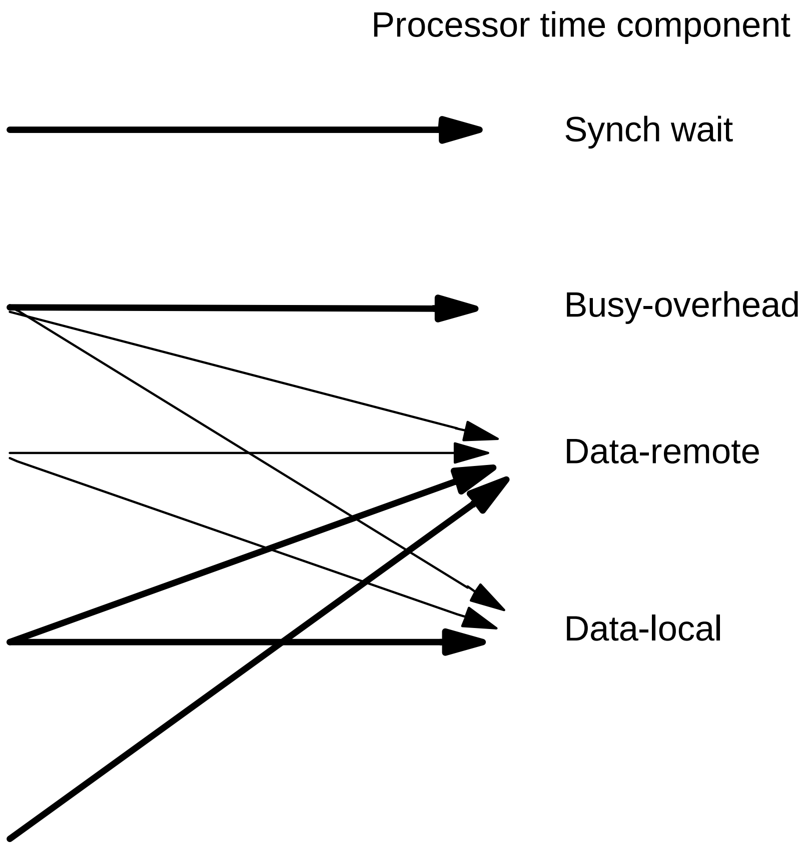


(a) Sequential

(b) Parallel with four processors

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



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

Parallel Application Case Studies

Examine Ocean and Barnes-Hut (others in book)

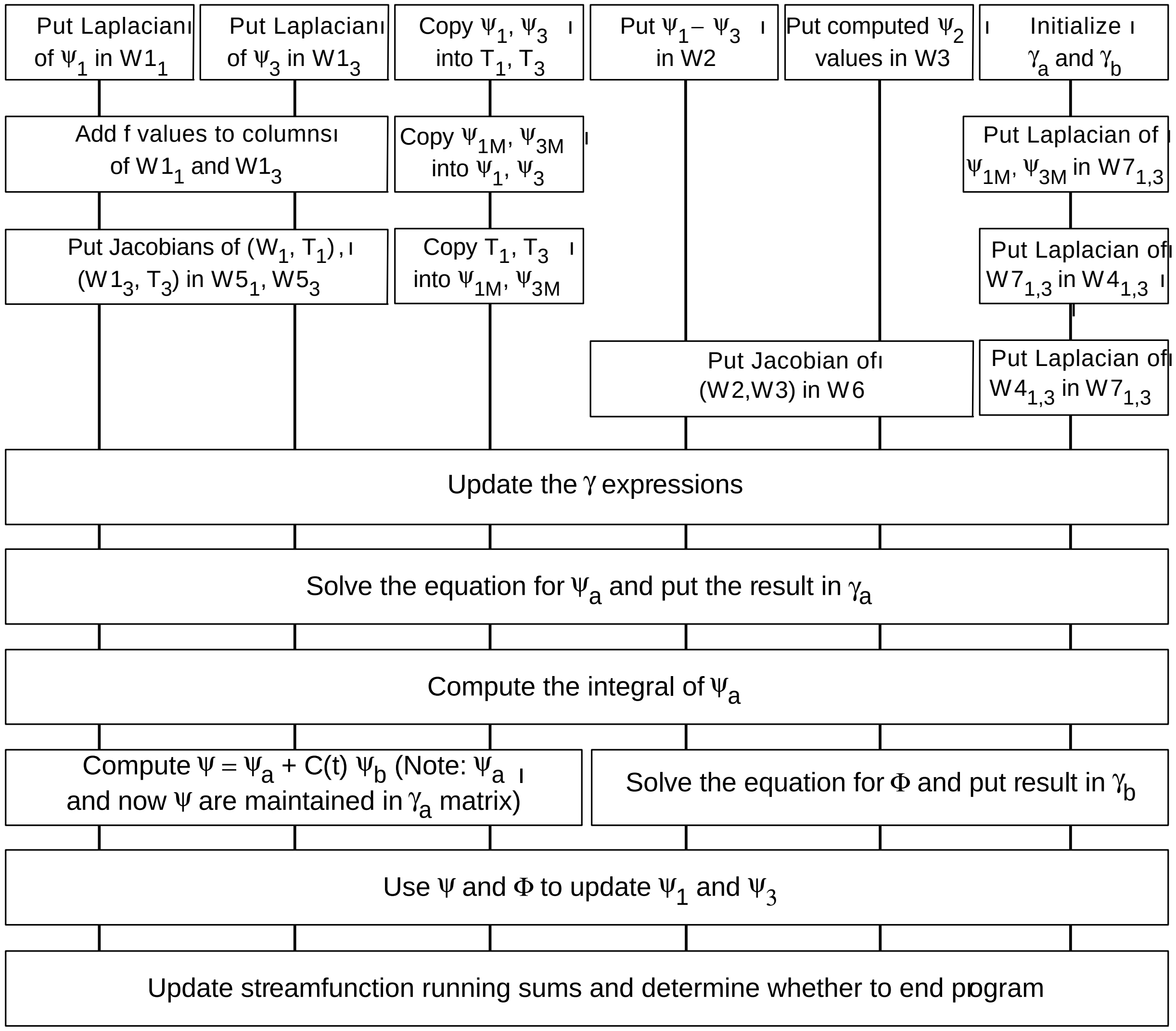
Assume cache-coherent shared address space

Five parts for each application

- Sequential algorithms and data structures
- Partitioning
- Orchestration
- Mapping
- Components of execution time on SGI Origin2000

Case Study 1: Ocean

Computations in a Time-step:


Put Laplacian of Ψ_1 in $W1_1$
Put Laplacian of Ψ_3 in $W1_3$
Copy Ψ_1, Ψ_3 into T_1, T_3
Put $\Psi_1 - \Psi_3$ in W2
Put computed Ψ_2 values in W3
Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$
Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3
Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put Jacobians of (W_1, T_1), ($W1_3$, T_3) in $W5_1$, $W5_3$
Copy T_1, T_3 into Ψ_{1M}, Ψ_{3M}
Put Laplacian of $W7_{1,3}$ in $W4_{1,3}$
Put Jacobian of (W2,W3) in W6
Put Laplacian of $W4_{1,3}$ in $W7_{1,3}$
Update the γ expressions
Solve the equation for Ψ_a and put the result in γ_a
Compute the integral of Ψ_a
Compute $\Psi = \Psi_a + C(t)\,\Psi_b$ (Note: Ψ_a and now Ψ are maintained in γ_a matrix)
Solve the equation for Φ and put result in γ_b
Use Ψ and Φ to update Ψ_1 and Ψ_3
Update streamfunction running sums and determine whether to end program

Partitioning

Exploit data parallelism

- Function parallelism only to reduce synchronization

Static partitioning within a grid computation

- Block versus strip
  - inherent communication versus spatial locality in communication
- Load imbalance due to border elements and number of boundaries

Solver has greater overheads than other computations

Orchestration and Mapping

Spatial Locality similar to equation solver

- Except lots of grids, so cache conflicts across grids

Complex working set hierarchy

- A few points for near-neighbor reuse, three subrows, partition of one grid, partitions of multiple grids…
- First three or four most important
- Large working sets, but data distribution easy

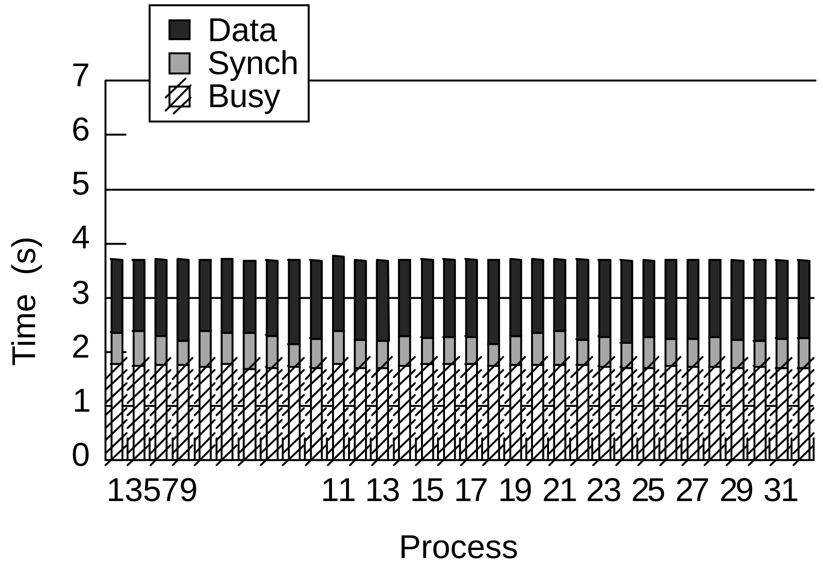
Synchronization

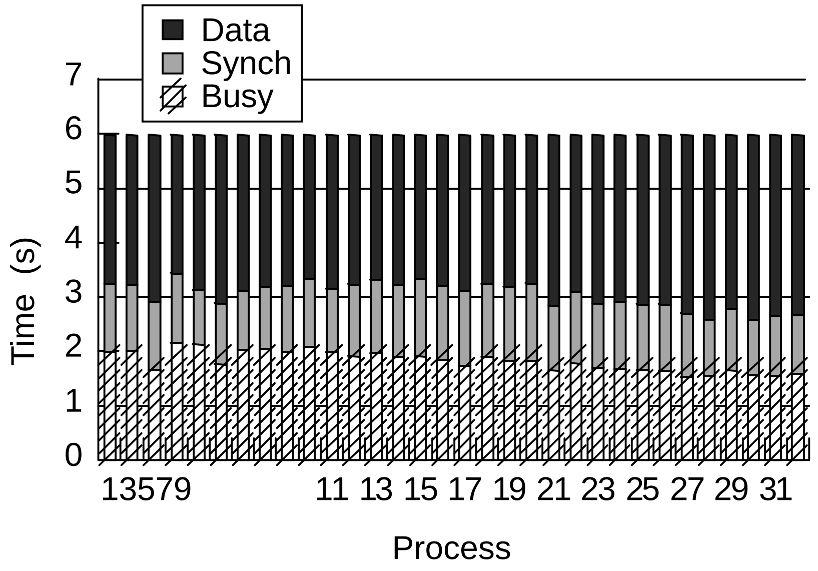
- Barriers between phases and solver sweeps
- Locks for global variables
- Lots of work between synchronization events

Mapping: easy mapping to 2-d array topology or richer

Execution Time Breakdown

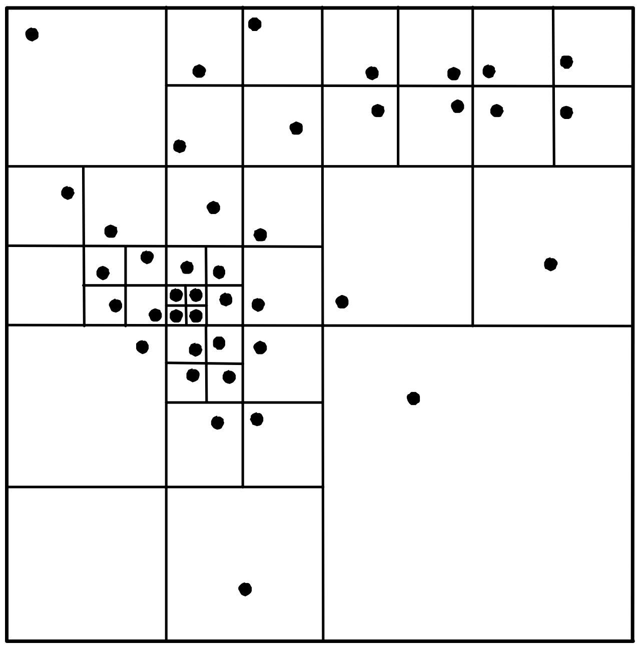
•1030 x 1030 grids with block partitioning on 32-processor Origin2000



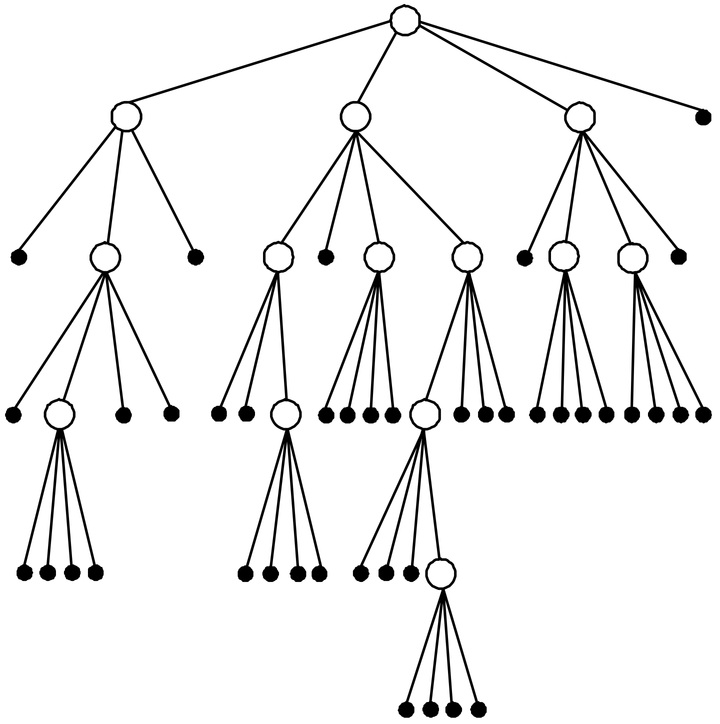


- 4-d grids much better than 2-d, despite very large caches on machine
  - data distribution is much more crucial on machines with smaller caches
- Major bottleneck in this configuration is time waiting at barriers
  - imbalance in memory stall times as well

Case Study 2: Barnes-Hut



(a) The spatial domain



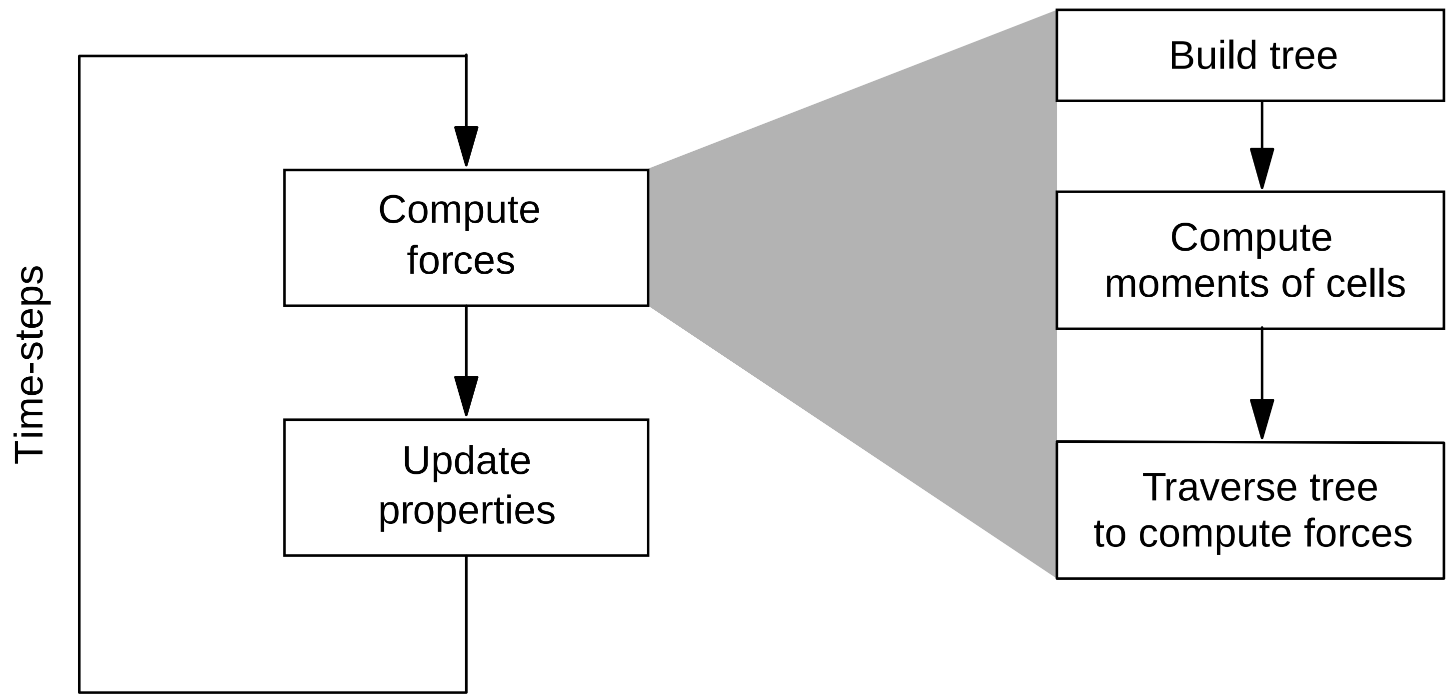
(b) Quadtree representation

Locality Goal:

- *Particles close together in space should be on same processor*

Difficulties: Nonuniform, dynamically changing

Application Structure



- 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

Partitioning

Decomposition: bodies in most phases, cells in computing moments

Challenges for assignment:

- Nonuniform body distribution => work and comm. Nonuniform
  - Cannot assign by inspection
- Distribution changes dynamically across time-steps
  - Cannot assign statically
- Information needs fall off with distance from body
  - Partitions should be spatially contiguous for locality
- Different phases have different work distributions across bodies
  - No single assignment ideal for all
  - Focus on force calculation phase
- Communication needs naturally fine-grained and irregular

Load Balancing

- Equal particles ≠ equal work.
  - Solution: Assign costs to particles based on the work they do

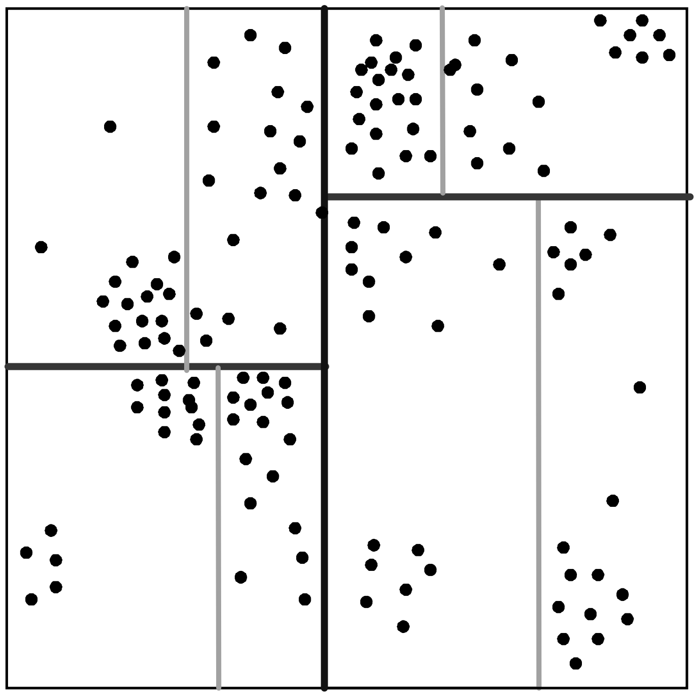
- Work unknown and changes with time-steps
  - Insight : System evolves slowly
  - Solution: *Count* work per particle, and use as cost for next time-step.

Powerful technique for evolving physical systems

A Partitioning Approach: ORB

Orthogonal Recursive Bisection:

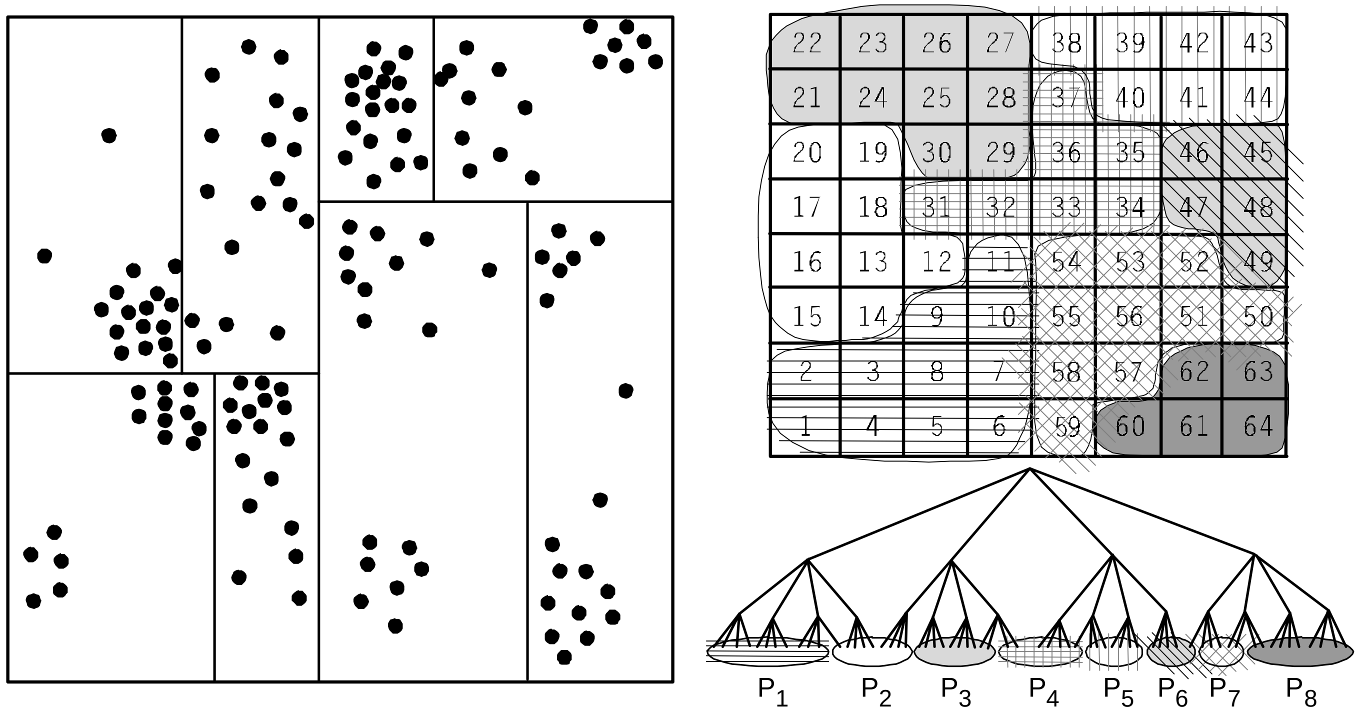
- Recursively bisect space into subspaces with equal work
  - Work is associated with bodies, as before
- Continue until one partition per processor



- High overhead for large no. of processors

Another Approach: Costzones

Insight: Tree already contains an encoding of spatial locality.

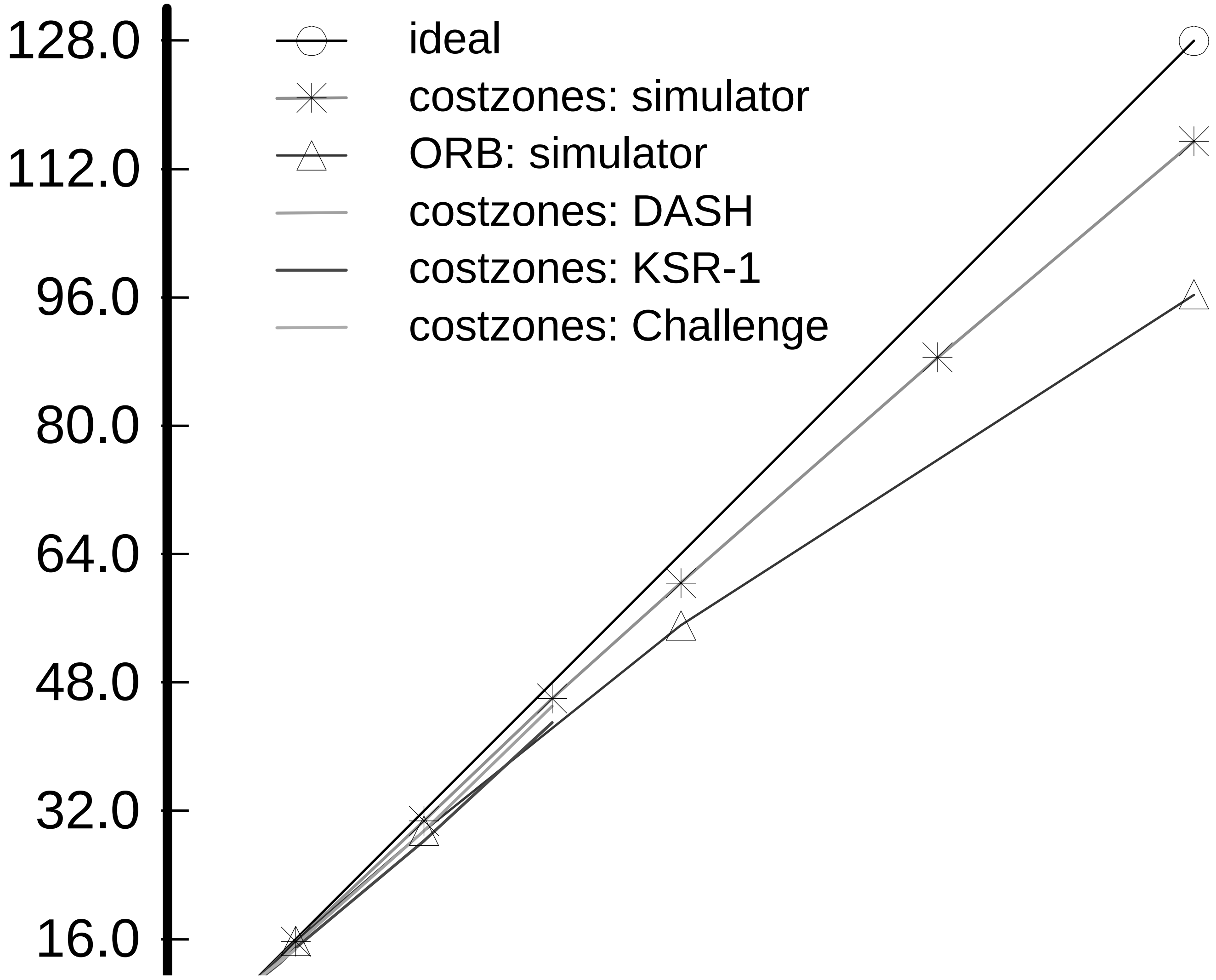


(a) ORB (b) Costzones

- Costzones is low-overhead and very easy to program

Performance Comparison

- Speedups on simulated multiprocessor (16K particles)
- Extra work in ORB partitioning is key difference



Orchestration and Mapping

Spatial locality: Very different than in Ocean, like other aspects

- Data distribution is much more difficult than
  - Redistribution across time-steps
  - Logical granularity (body/cell) much smaller than page
  - Partitions contiguous in physical space does not imply contiguous in array
  - But, good temporal locality, and most misses logically non-local anyway
- Long cache blocks help within body/cell record, not entire partition

Temporal locality and working sets:

- Important working set scales as $1/\theta^2 \log n$
- Slow growth rate, and fits in second-level caches, unlike Ocean

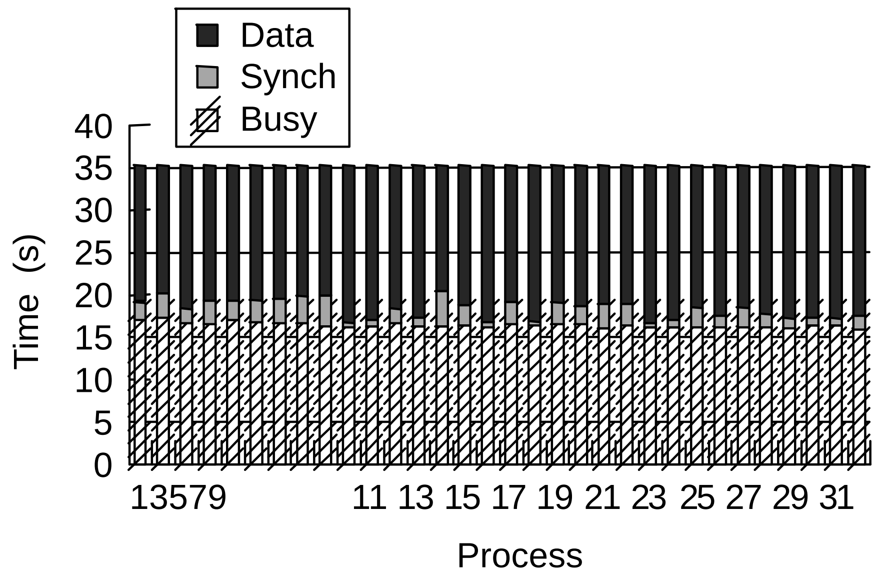
Synchronization:

- Barriers between phases
- No synch within force calculation: data written different from data read
- Locks in tree-building, pt. to pt. event synch in center of mass phase

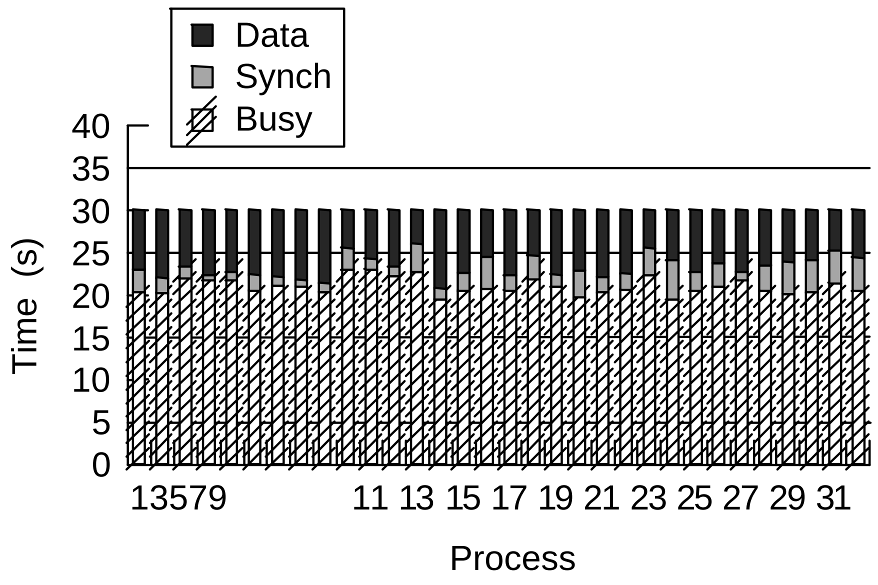
Mapping: ORB maps well to hypercube, costzones to linear array

Execution Time Breakdown

- 512K bodies on 32-processor Origin2000
  - Static, quite randomized in space, assignment of bodies versus costzones



(a) Static assignment of bodies



(b) Semistatic costzone assignment

- Problem with static case is communication/locality, not load balance!

Raytrace

Rays shot through pixels in image are called *primary rays*

- Reflect and refract when they hit objects
- Recursive process generates ray tree per primary ray

Hierarchical spatial data structure keeps track of primitives in scene

- Nodes are space cells, leaves have linked list of primitives

Tradeoffs between execution time and image quality

Partitioning

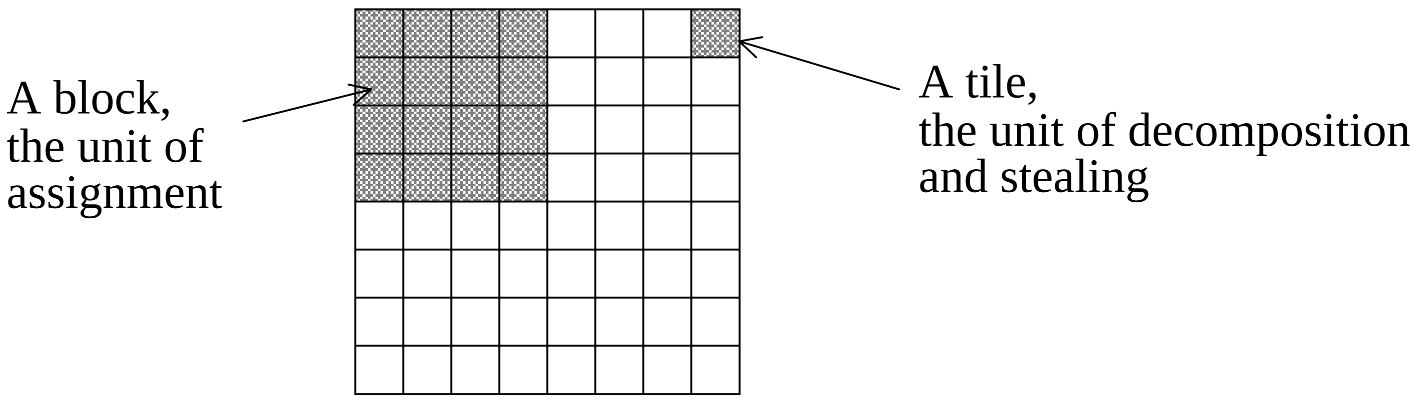
Scene-oriented approach

- Partition scene cells, process rays while they are in an assigned cell

Ray-oriented approach

- Partition primary rays (pixels), access scene data as needed
- Simpler; used here

Need dynamic assignment; use contiguous blocks to exploit spatial coherence among neighboring rays, plus tiles for task stealing



Could use 2-D interleaved (scatter) assignment of tiles instead

Orchestration and Mapping

Spatial locality

- Proper data distribution for ray-oriented approach very difficult
- Dynamically changing, unpredictable access, fine-grained access
- Better spatial locality on image data than on scene data
  - Strip partition would do better, but less spatial coherence in scene access

Temporal locality

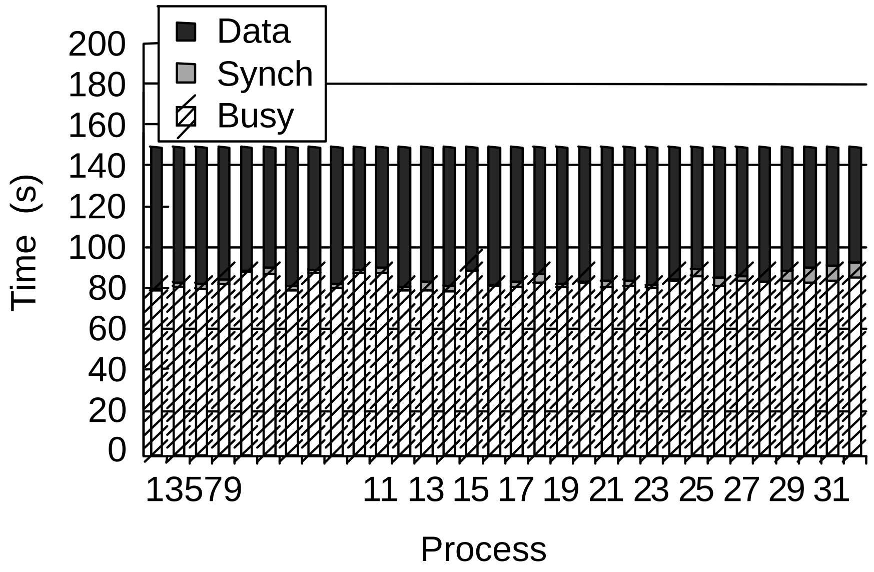
- Working sets much larger and more diffuse than Barnes-Hut
- But still a lot of reuse in modern second-level caches
  - SAS program does not replicate in main memory

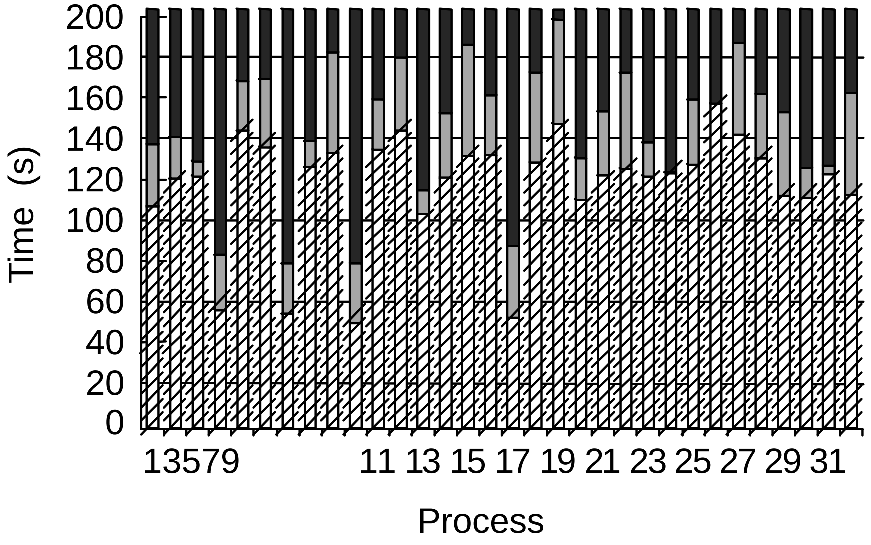
Synchronization:

- One barrier at end, locks on task queues

Mapping: natural to 2-d mesh for image, but likely not important

Execution Time Breakdown





- Task stealing clearly very important for load balance

Implications for Programming Models

Shared address space and explicit message passing

- SAS may provide coherent replication or may not
- Focus primarily on former case

Assume distributed memory in all cases

Recall any model can be supported on any architecture

- Assume both are supported efficiently
- Assume communication in SAS is only through loads and stores
- Assume communication in SAS is at cache block granularity

Issues to Consider

Functional issues:

- Naming
- Replication and coherence
- Synchronization

Organizational issues:

- Granularity at which communication is performed

Performance issues

- Endpoint overhead of communication
  - (latency and bandwidth depend on network so considered similar)
- Ease of performance modeling

Cost Issues

- Hardware cost and design complexity

Naming

SAS: similar to uniprocessor; system does it all

MP: each process can only directly name the data in its address space

- Need to specify from where to obtain or where to transfer nonlocal data
- Easy for regular applications (e.g. Ocean)
- Difficult for applications with irregular, time-varying data needs
  - Barnes-Hut: where the parts of the tree that I need? (change with time)
  - Raytrace: where are the parts of the scene that I need (unpredictable)
- Solution methods exist
  - Barnes-Hut: Extra phase determines needs and transfers data before computation phase
  - Raytrace: scene-oriented rather than ray-oriented approach
  - both: emulate application-specific shared address space using hashing

Replication

Who manages it (i.e. who makes local copies of data)?

- SAS: system, MP: program

Where in local memory hierarchy is replication first done?

- SAS: cache (or memory too), MP: main memory

At what granularity is data allocated in replication store?

- SAS: cache block, MP: program-determined

How are replicated data kept coherent?

- SAS: system, MP: program

How is replacement of replicated data managed?

- SAS: dynamically at fine spatial and temporal grain (every access)
- MP: at phase boundaries, or emulate cache in main memory in software

Of course, SAS affords many more options too (discussed later)

Amount of Replication Needed

Mostly local data accessed => little replication

Cache-coherent SAS:

- Cache holds active working set
  - replaces at fine temporal and spatial grain (so little fragmentation too)
- Small enough working sets => need little or no replication in memory

Message Passing or SAS without hardware caching:

- Replicate all data needed in a phase in main memory
  - replication overhead can be very large (Barnes-Hut, Raytrace)
  - limits scalability of problem size with no. of processors
- Emulate cache in software to achieve fine-temporal-grain replacement
  - expensive to manage in software (hardware is better at this)
  - may have to be conservative in size of cache used
  - fine-grained message generated by misses expensive (in message passing)
  - programming cost for cache and coalescing messages

Communication Overhead and Granularity

Overhead directly related to hardware support provided

- Lower in SAS (order of magnitude or more)

Major tasks:

- Address translation and protection
  - SAS uses MMU
  - MP requires software protection, usually involving OS in some way
- Buffer management
  - fixed-size small messages in SAS easy to do in hardware
  - flexible-sized message in MP usually need software involvement
- Type checking and matching
  - MP does it in software: lots of possible message types due to flexibility
- A lot of research in reducing these costs in MP, but still much larger

Naming, replication and overhead favor SAS

- Many irregular MP applications now emulate SAS/cache in software

Block Data Transfer

Fine-grained communication not most efficient for long messages

- Latency and overhead as well as traffic (headers for each cache line)

SAS: can using block data transfer

- Explicit in system we assume, but can be automated at page or object level in general (more later)
- Especially important to amortize overhead when it is high
  - latency can be hidden by other techniques too

Message passing:

- Overheads are larger, so block transfer more important
- But very natural to use since message are explicit and flexible
  - Inherent in model

Synchronization

SAS: Separate from communication (data transfer)

- Programmer must orchestrate separately

Message passing

- Mutual exclusion by fiat
- Event synchronization already in send-receive match in synchronous
  - need separate orchestratation (using probes or flags) in asynchronous

Hardware Cost and Design Complexity

Higher in SAS, and especially cache-coherent SAS

But both are more complex issues

- Cost
  - must be compared with cost of replication in memory
  - depends on market factors, sales volume and other nontechnical issues
- Complexity
  - must be compared with complexity of writing high-performance programs
  - Reduced by increasing experience

Performance Model

Three components:

- Modeling cost of primitive system events of different types
- Modeling occurrence of these events in workload
- Integrating the two in a model to predict performance

Second and third are most challenging

Second is the case where cache-coherent SAS is more difficult

- replication and communication implicit, so events of interest implicit
  - similar to problems introduced by caching in uniprocessors
- MP has good guideline: messages are expensive, send infrequently
- Difficult for irregular applications in either case (but more so in SAS)

Block transfer, synchronization, cost/complexity, and performance modeling advantageus for MP

Summary for Programming Models

Given tradeoffs, architect must address:

- Hardware support for SAS (transparent naming) worthwhile?
- Hardware support for replication and coherence worthwhile?
- Should explicit communication support also be provided in SAS?

Current trend:

- Tightly-coupled multiprocessors support for cache-coherent SAS in hw
- Other major platform is clusters of workstations or multiprocessors
  - currently don’t support SAS in hardware, mostly use message passing

Summary

Crucial to understand characteristics of parallel programs

- Implications for a host or architectural issues at all levels

Architectural convergence has led to:

- Greater portability of programming models and software
  - Many performance issues similar across programming models too
- Clearer articulation of performance issues
  - Used to use PRAM model for algorithm design
  - Now models that incorporate communication cost (BSP, logP,….)
  - Emphasis in modeling shifted to end-points, where cost is greatest
  - But need techniques to model application behavior, not just machines

Performance issues trade off with one another; iterative refinement

Ready to understand using workloads to evaluate systems issues