Parallel Programming (for Performance)

Chapter 3 of Culler & Singh

- **Introduction**
- Load Balance and Synchronization Wait Time
- Reducing Inherent Communication
- Reducing Extra Work
- Applications

Introduction

- Examine the key factors that limit parallel performance
  - How are they addressed in a wide range of problems?
- Interdependencies of program and system
  - More complicated + important on multicore (vs 1 CPU)
- Focus on software techniques for performance

Partitioning for Performance

- Assumptions
  - Machine is a set of cooperating processors
  - Communication between them is expensive

- Three optimizations goals
  1. Balance workloads and reduce sync time
  2. Reduce communication between processors
  3. Reduce the extra work done

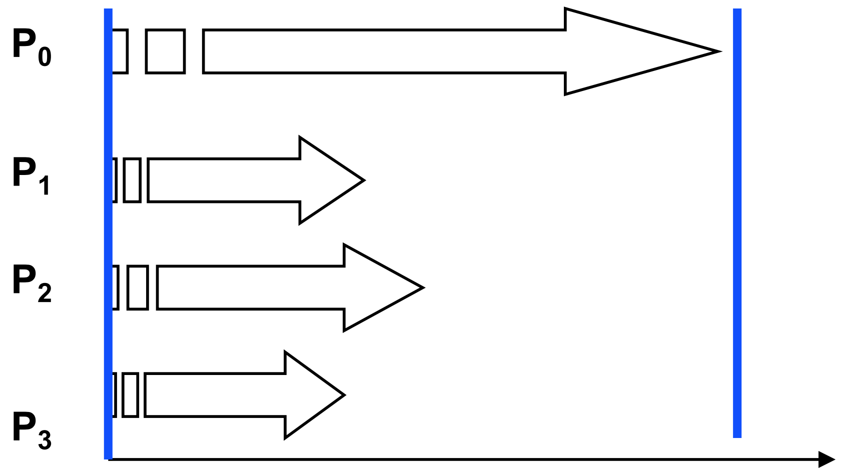
Partitioning for Performance

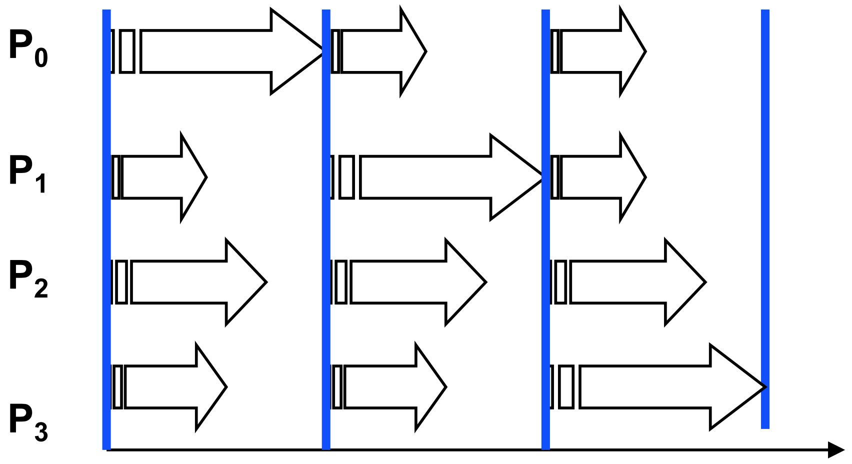
- These goals are at the odds with one another
  - Minimum communication with 1 CPU
  - Ultimate load imbalance
- On the other hand
  - Best load balance requires a lot of communication
- Find right balance / compromise

Load Balance and Synchronization Wait Time

- Balanced workload
  - Ensure every processor does the same amount of work *at the same time*
- Max speedup

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





Load Balance and Synchronization Wait Time

- **Process of balancing consist of four parts**
  1. Identify enough concurrency (decomposition)
  2. How to manage concurrency (static or dynamic)
  3. Determine the granularity of concurrency
  4. Reduce serialization and synchronization

Identifying Concurrency

- Data parallelism
  - Loops often apply similar operation sequences on elements of large data structure
- Functional parallelism
  - Different calculations on same or different data
  - Another form of pipelining
- Often both available together
  - Hierarchy of parallelism

Identifying Concurrency

- Data parallelism
  - Grows with the data set size
- Functional parallelism
  - Doesn’t grow with data
  - Difficult to balance
    - Functions usually require different amount of work, have different scaling characteristics

Managing Concurrency

- Assignment
  - Group tasks into processes
- Key issue
  - Can a static (predetermined) assignment achieve good load balance?
  - Or is a dynamic approach necessary?
- Each method has pros + cons

Static Load Balancing

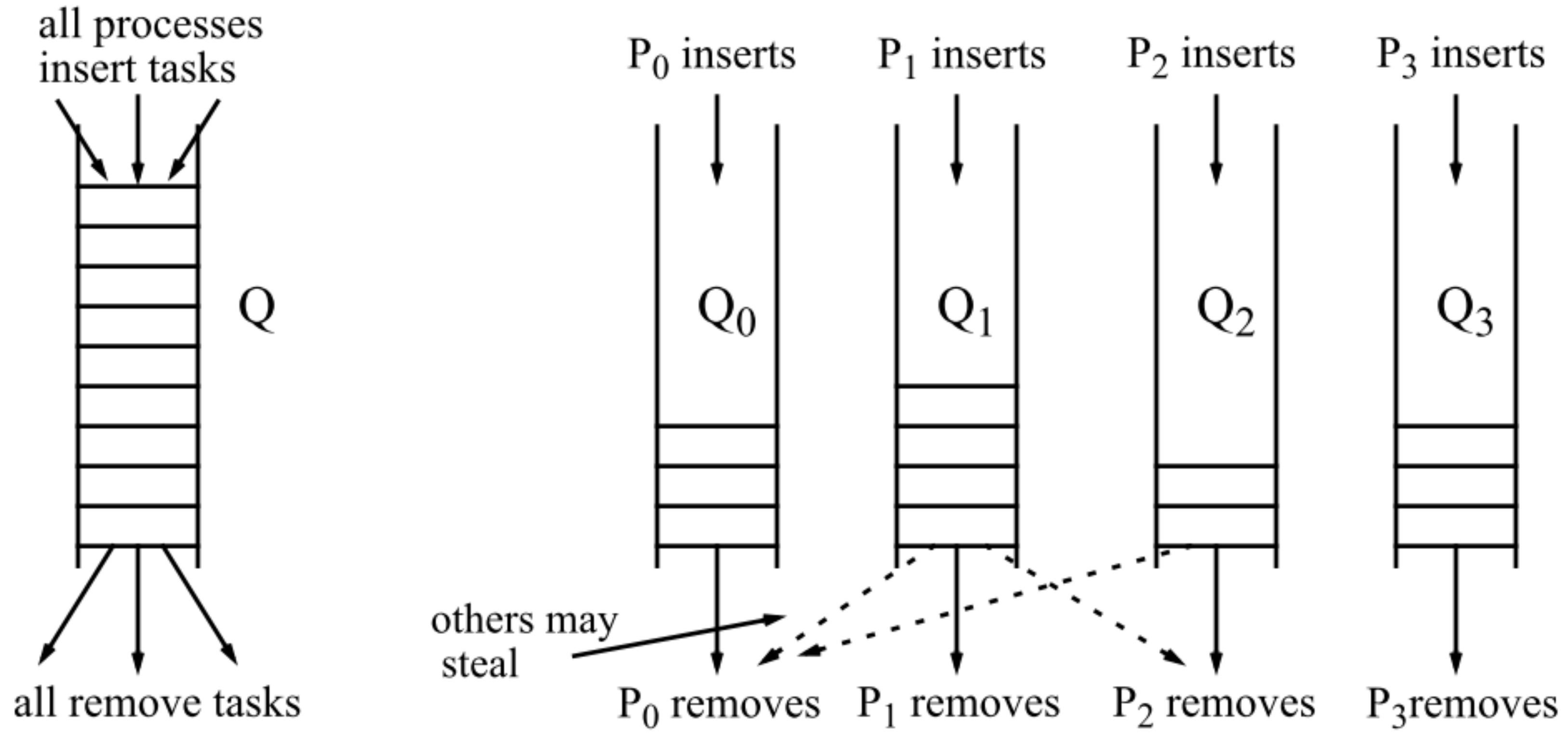
- Pros
  - Assignment is determined once
  - Low task management overhead at runtime

- Cons
  - Workload per process must be predictable
  - Environmental conditions must not disturb system
    - Eg, interference from other applications slows down some processors, but not others

Dynamic Load Balancing

- Pros
  - Adapts load imbalance at runtime, achieves good balance
- Cons
  - Adds task management overhead, may run continuously
- Two variants:
  - Semi-static: workload balance changes slowly
    - Initial assignment, recomputed periodically, eg Barnes-Hut
  - Dynamic: workload balance unpredictable / changes rapidly
    - Pool of tasks, take and add tasks until done
    - Unpredictable program or environment
    - Requires continuously redistributed workload, eg Raytrace

Dynamic Load Balance: Task Queues



(a) Centralized task queue

(b) Distributed task queues (one per process)

Task Queues

- Centralized
  - Simple and good for large tasks
  - May increase communication
  - Tasks too large → workload imbalance
  - Many processors → contention for queue access
- Distributed
  - Local, each process assigned a group of tasks
  - Task stealing from other queues
    - Must search many queues for new work

Granularity of Tasks

- Task granularity
  - Amount of work associated with a task
  - # instructions, execution time
- Fine-grained
  - Better load balance
  - More tasks to divide => more concurrency
  - Overhead of generating/selecting/managing tasks
- Coarse-grained
  - Often less load balance

Determining the Granularity of Tasks

- Task Granularity with Dynamic Task Queuing
  - Large tasks have lower overhead
    - Balance work better by prioritizing long-running tasks
  - Smaller tasks access queues more frequently, especially centralized queues
- Task Granularity with Static Assignment
  - Granularity has less influence with static assignment
  - Communication and contention are affected by the mapping to processors, not granularity

Reducing Synchronization

- Synchronization = Serialization
- To reduce, be careful how we assign tasks
- Fine-grained vs. coarse-grained
  - Eg: lock access to a data element vs entire matrix
  - Coarse-grained: simple, fewer resources, less parallelism
  - Fine-grained: lowers contention, more complexity, more resources, more parallelism

Reducing Synchronization

- Event synchronization / serialization
  - Reduce use of conservative synchronization
    - e.g. use a point-to-point sync instead of global barrier
  - Use fine-grained granularity
    - More sync operations ➔ program complexity, correctness
- Mutual exclusion
  - Separate locks for separate data
    - e.g. locking records in a database: lock per process, record, or field
    - Finer grain => less contention/serialization, more space overhead
  - Smaller, less frequent critical sections
    - Don’t do “extra reading” in critical section,
      only essential “read-modify-write”

Example

- Eg: central task queue
  - Multiple processes must add task, search for new task, or delete task
- Fine-grain or coarse-grain?
- Two locks
  - Insertion Lock
  - Deletion Lock
  - Search (read only) is outside of critical section, no locks

Implication of Load Balance

- Extend to reflect
  - Load imbalance
  - Time spent waiting at synchronization points

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

Final Words on Load Balancing

- In general it is the software responsibility

- Hardware can help
  – Support load balancing techniques as task stealing
  – Make it easy to access logically shared data
  – Efficient synchronization

- Introduction
- Load Balance and Synchronization Wait Time
- **Reducing Inherent Communication**
- Reducing Extra Work
- Applications

Reducing Inherent Communication

- Load balance 🡐 trade off 🡒 inter-processor communication
- Focus on reducing inherent communication
  - Process i provides data value needed by Process j
  - While preserving load balance

Reducing Inherent Communication

Communication is expensive!

$$\text{Measure: } \frac{\text{Sequential Work}}{\textit{Max}\text{ Work on any Processor}}$$

Communication determined by assignment of tasks to processes

- Tasks that access same data assigned to same process

Solving communication / load balance NP-hard in general

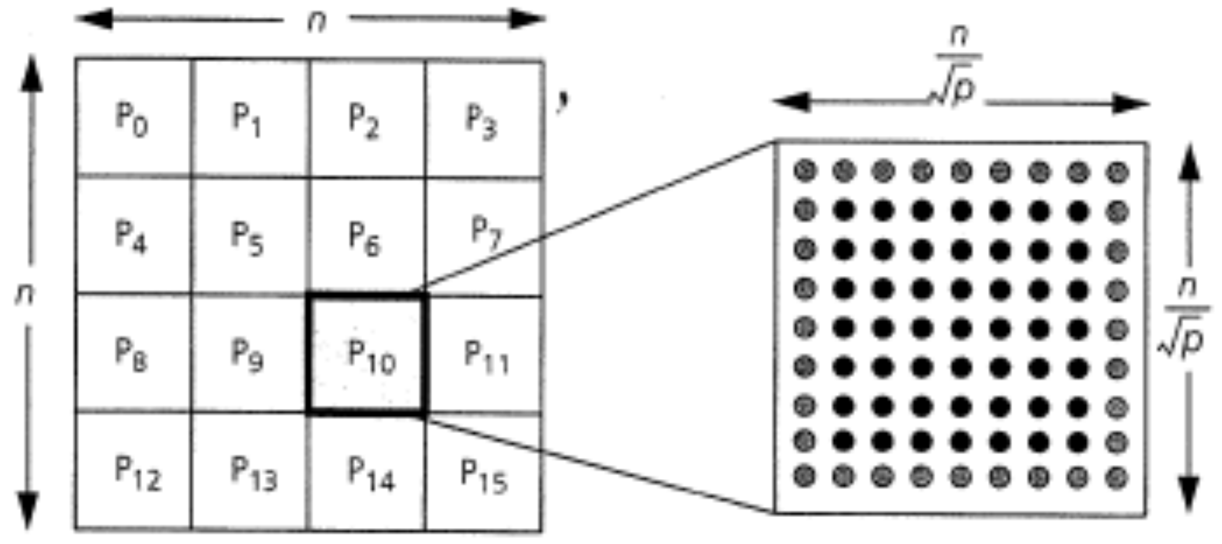
- Simple heuristic solutions work well in practice.

Reducing Inherent Communication

- Measure impact of communication
  - Communication-to-computation ratio
- Can be computed
  - Per process, or
  - Accumulated over all processes

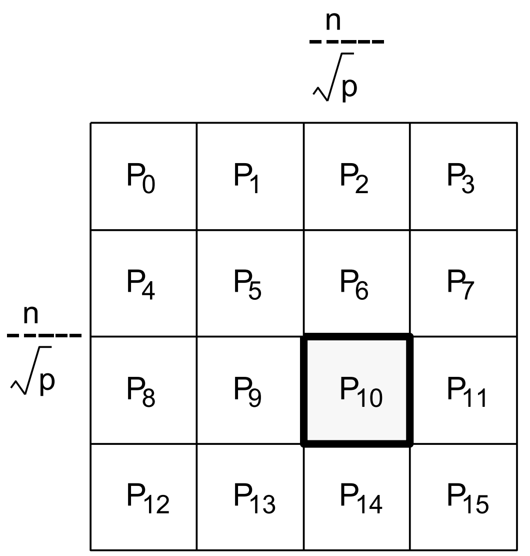
Example

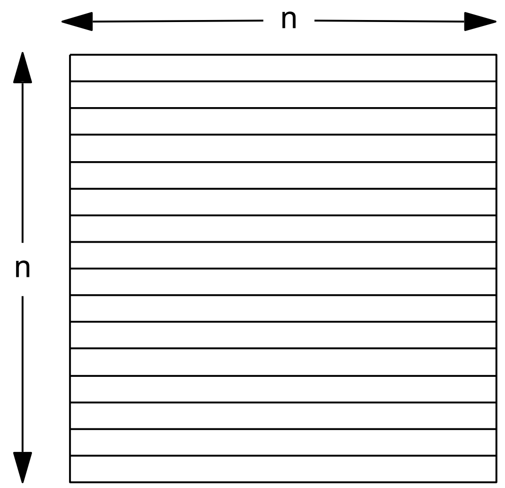
- The equation solver kernel



- Comm-to-comp ratio?
  - Communication: perimeter approx = 4n/sqrt(p)
  - Computation: area approx= n^2/p
  - Ratio = (4n/sqrt(p)) / (n^2/p) = 4sqrt(p) / n
  - Good: ratio goes to 0 (because n grows faster than p)

Example: tiles vs strips





- Comm to comp: $\frac{4*\sqrt{p}}{n}$ for tiles $\frac{2*p}{n}$ for strips
  - Application dependent: strip may be better in some cases
- Best domain decomposition depends on information requirements

Implications of Comm-to-Comp Ratio

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

- Need to keep communication balanced across processors as well

- Introduction
- Load Balance and Synchronization Wait Time
- Reducing Inherent Communication
- **Reducing Extra Work**
- Applications

Reducing Extra Work

- Extra work not found in serial version
- Common sources of extra work
  - Using redundant computation to avoid communication
  - Computing a good partition
  - Task, data and process management overhead
  - Imposing structure on communication
- Implications

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

Reducing Extra Work

- Common sources of extra work:
  - Using redundant computation to avoid communication
  - Computing a good partition
  - Task, data and process management overhead
  - Imposing structure on communication

- Architectural Implications:

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

What is a Multiprocessor?

A collection of cooperating processors

balance load, reduce inherent communication and extra work

A multi-cache, multi-memory system

- Role of these components essential, regardless of programming model
- Most of remaining performance issues focus on memory

Memory-oriented view

- Multiprocessor is Extended Memory Hierarchy
  - as seen by a processor
- Levels in Extended Memory 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
  - Cache effects
- Performance of a system = time needed to complete a program

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

- 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: what about caches of other processors?
- Distributed Memory: some local, some remote
- Management of levels
    - Registers are managed by the compiler
    - Caches managed transparently by hardware
    - Main memory depends on programming model
        - Message passing: explicit
        - Shared Address Space (SAS): implicit/transparent

Communication and Replication

Communication induced by finite capacity

- Like cache size and miss rate, or memory traffic in uniprocessors

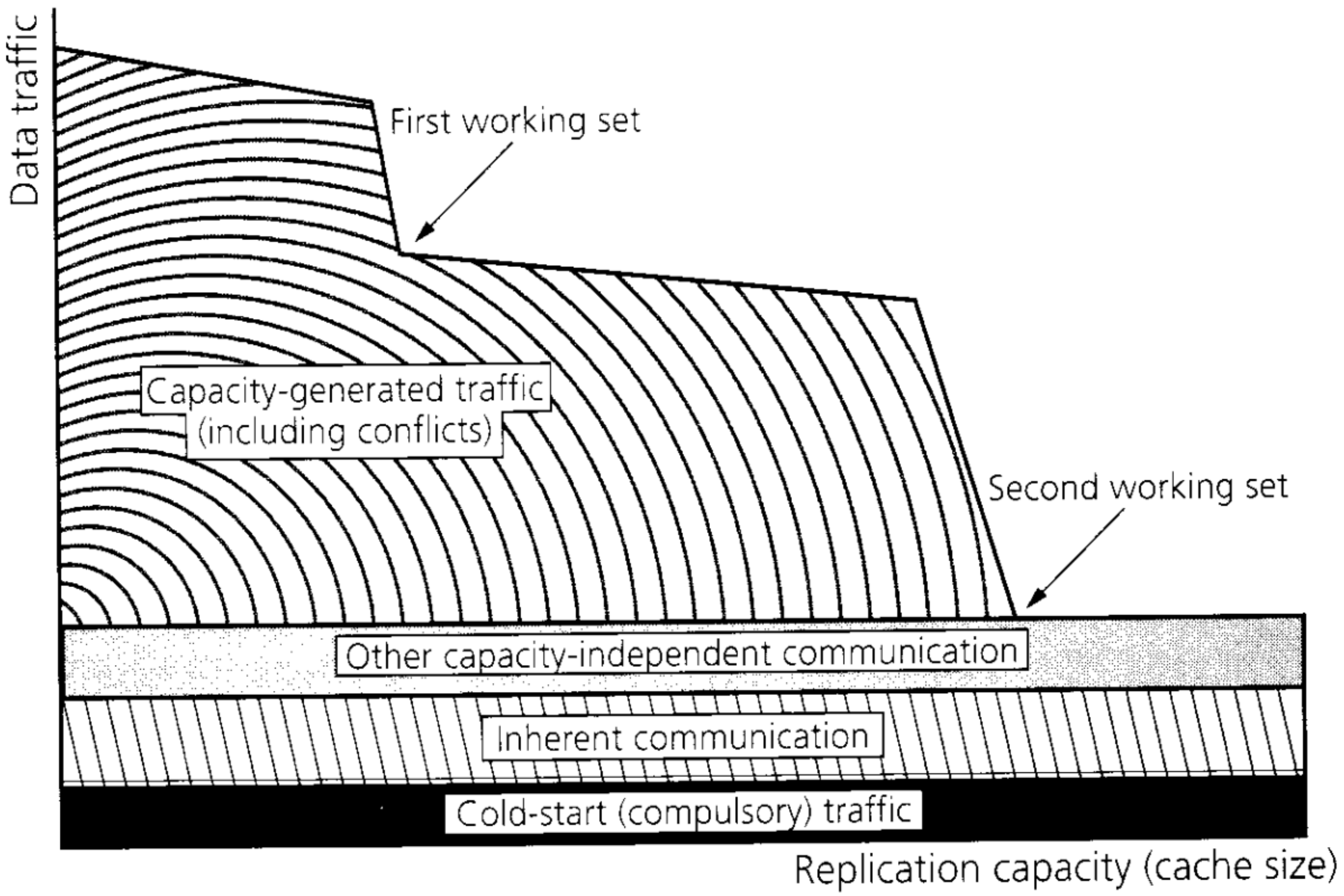
View as three level hierarchy for simplicity

- Local cache, local memory, remote memory (ignore network topology)

Classify “misses” in “cache”, extending uniprocessor view

- Compulsory or cold misses
  - First-touch of data, cannot avoid, even with a bigger cache
- Capacity misses
  - Hits in cache if cache is made bigger
- Conflict or collision misses
  - Hits in cache if cache is fully associative
- Communication or coherence misses
  - Hits in cache if it wasn’t modified/removed by another processor

Caches: Working Set Perspective



Orchestration for Performance

Reducing amount of communication

- Exploit spatial, temporal locality in extended hierarchy
  - Techniques often similar to those on uniprocessors
- Change logical data sharing patterns in algorithm

Structuring communication to reduce cost

Let’s examine techniques for both...

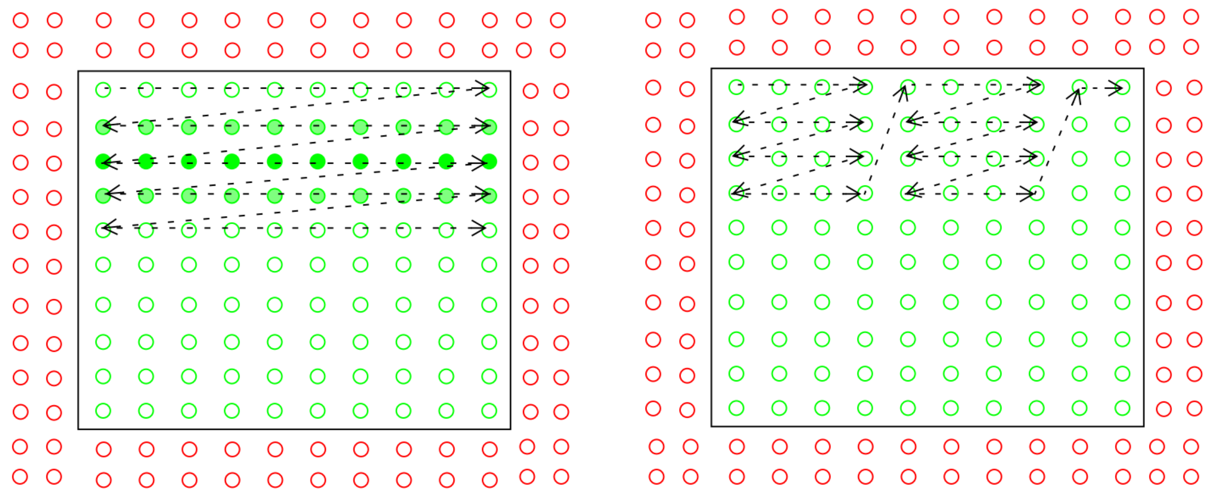
Exploiting Temporal Locality

Structure algorithm so working sets map well to hierarchy

- assigning tasks that tend to access the same data to the same process
- schedule tasks for data reuse once assigned

Example

- Kernel solver



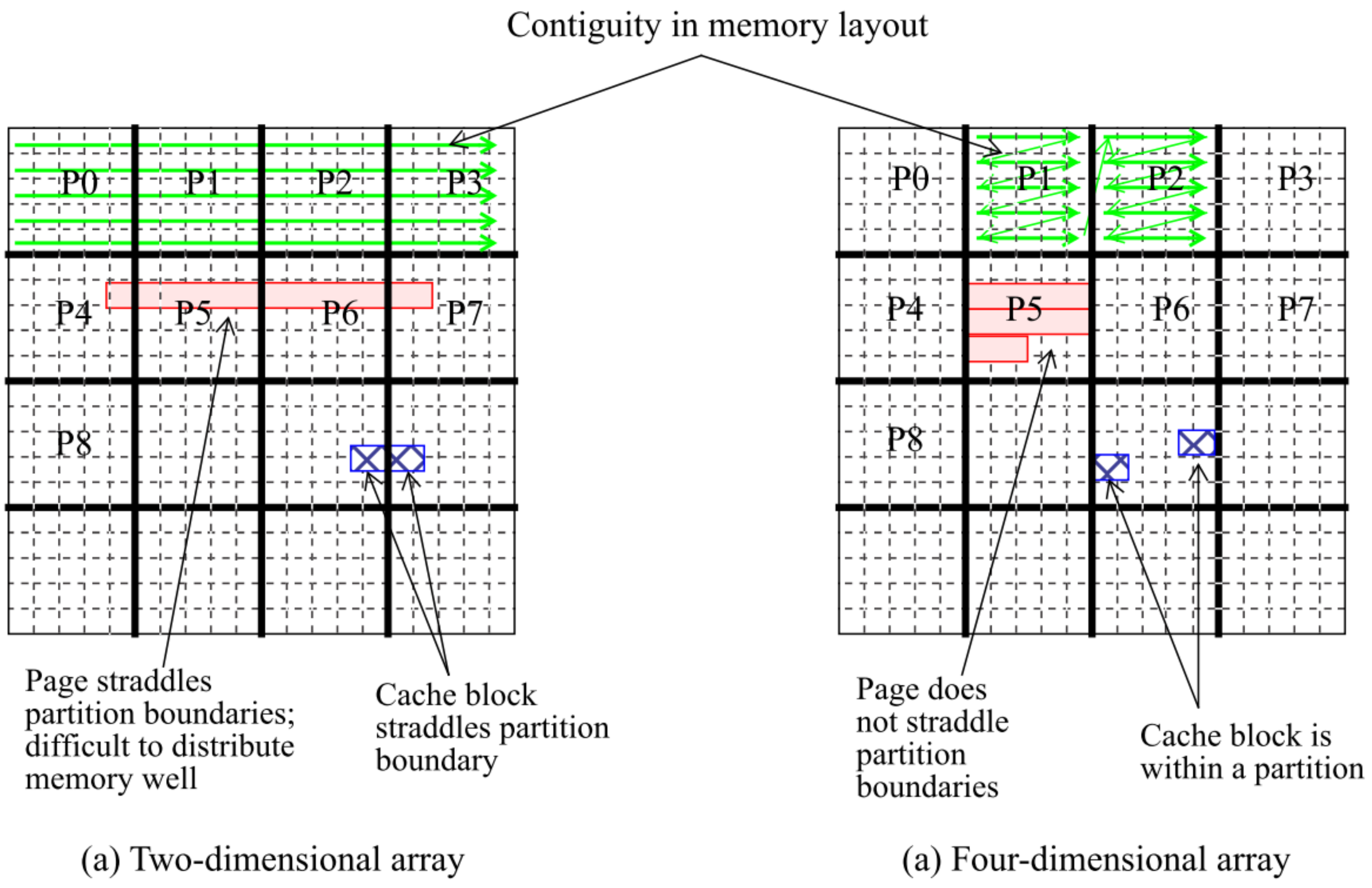
(a) Unblocked access pattern in a sweep (b) Blocked access pattern with B=4

- Most useful when O(n^{k+1}) operations on O(n^k) data

Exploiting Spatial Locality Example

Besides capacity, granularities are important:

- Granularity of allocation (virtual memory page size, eg 4kB)
- Granularity of coherence (cache blocks, eg 64B)



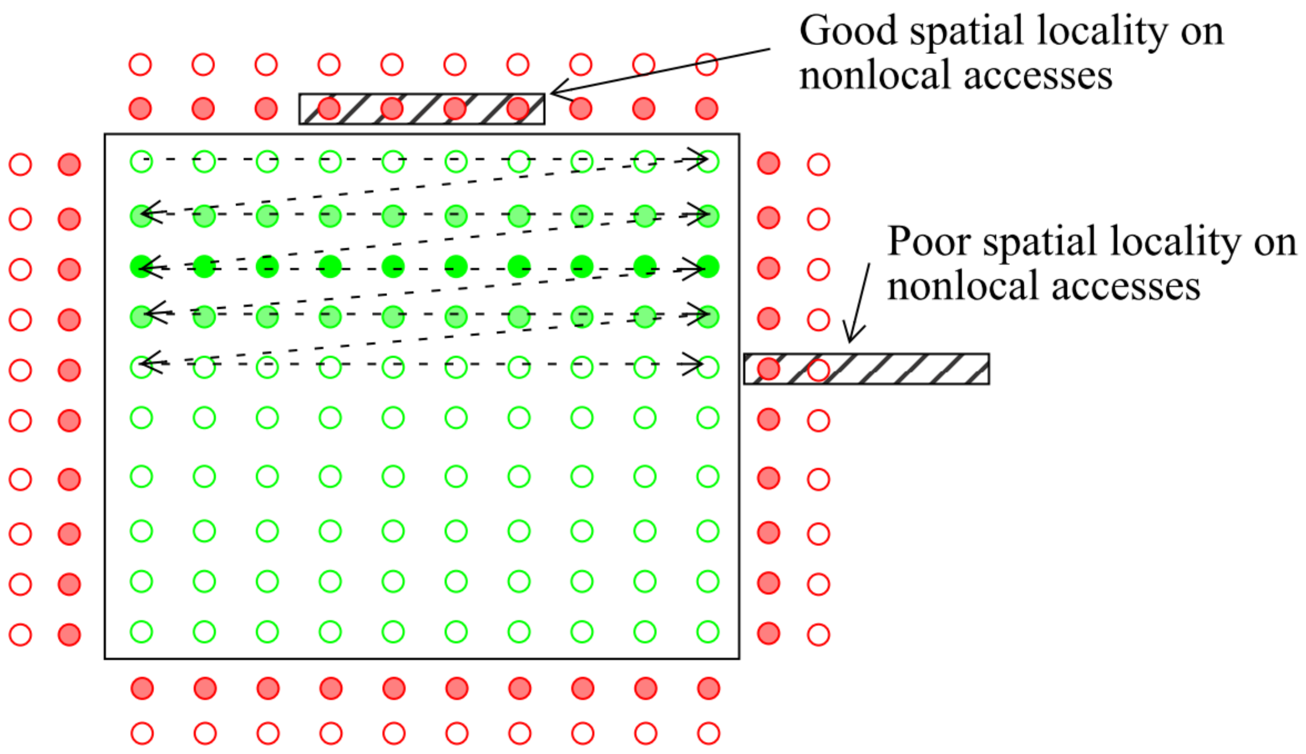
Contiguity in memory layout
P0
P1
P2
P3
P4
P5
P6
P7
P8
Page straddles partition boundaries; difficult to distribute memory well
Cache block straddles partition boundary
Page does not straddle partition boundaries
Cache block is within a partition

(a) Two-dimensional array

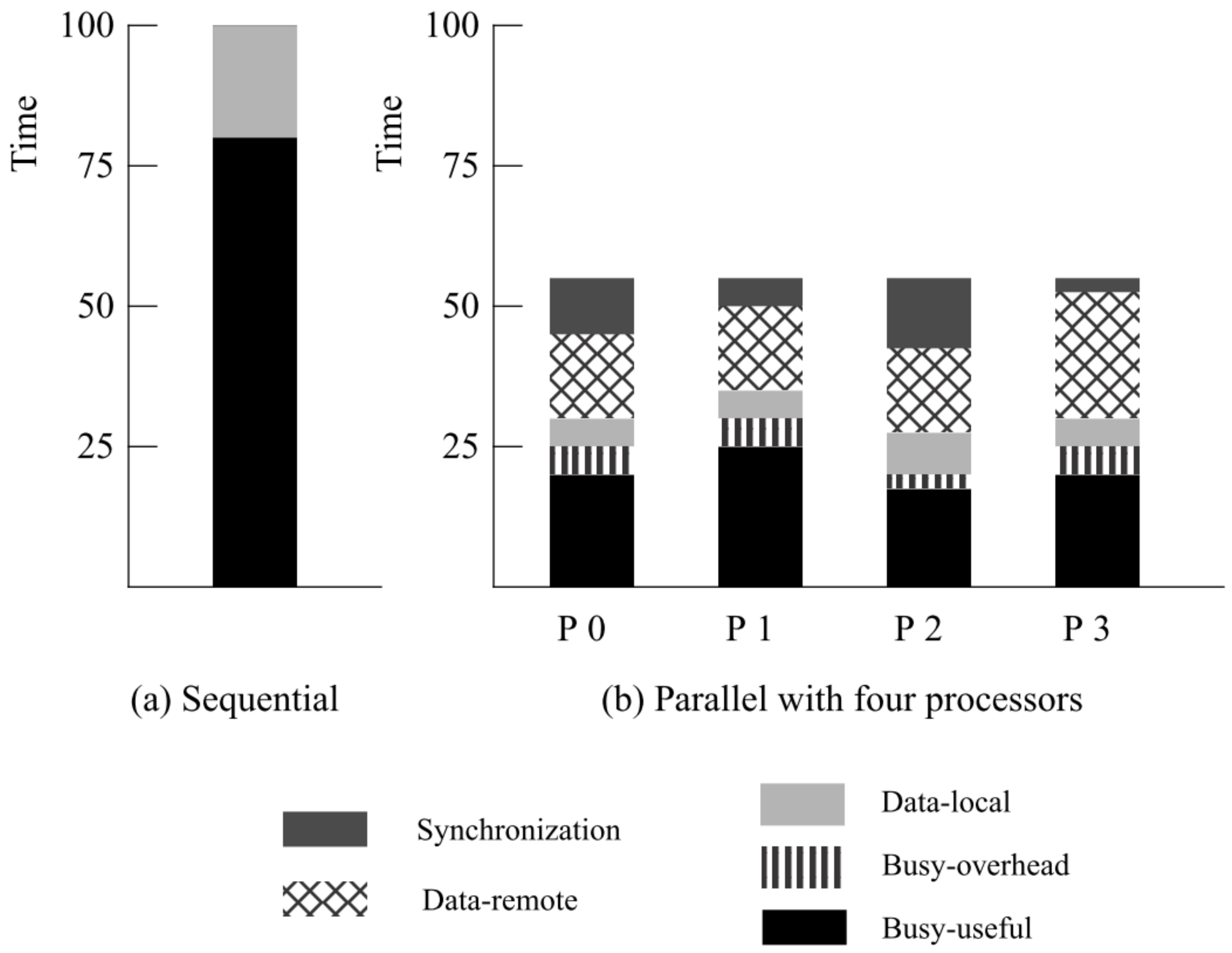
(a) Four-dimensional array

Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows


Good spatial locality on nonlocal accesses
Poor spatial locality on nonlocal accesses

Processor-Centric Performance Perspective



Summary: Processor-Centric

$$\text{Speedup} \leq \frac{\text{Busy(l) + Data(l)}}{\text{BusyUsful(p) + DataLocal(P) +Synch(p)+DateRemote(P)+BusyOverhead(P)}}$$

- Goal is to reduce denominator components
  - Primarily responsibility of programmer
  - Architecture cannot directly improve load imbalance or communication
- Architecture cannot support more efficient communication and synchronization
  - Overlapping communication + computation
  - Support for barriers or locks
  - Low-latency messaging for short messages
  - High-bandwidth block-transfer for long messages

Synchronization

Message passing synchronization

- Mutual exclusion is automatically provided
- Implicit event synchronization in synchronous send-receive pair
  - May need separate orchestration (using probes or flags) in asynchronous portions

SAS (shared address space) sychronization

- Separate from communication (data transfer)
- Programmer must orchestrate separately

Block Data Transfer

Fine-grained communication not efficient for long messages

- Latency, congestion, overhead

SAS: use block data copy

- Explicit, or can be automated at page or object level

Message passing:

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

Summary

- Crucial to understand characteristics of parallel programs

- Need to understand overhead issues

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

$$\text{Speedup} \leq \frac{\text{Busy(l)} + \text{Data(l)}}{\text{BusyUseful(p)} + \text{DataLocal(P)} + \text{Sync(p)} + \text{DateRemote(P)} + \text{BusyOverhead(P)}}$$

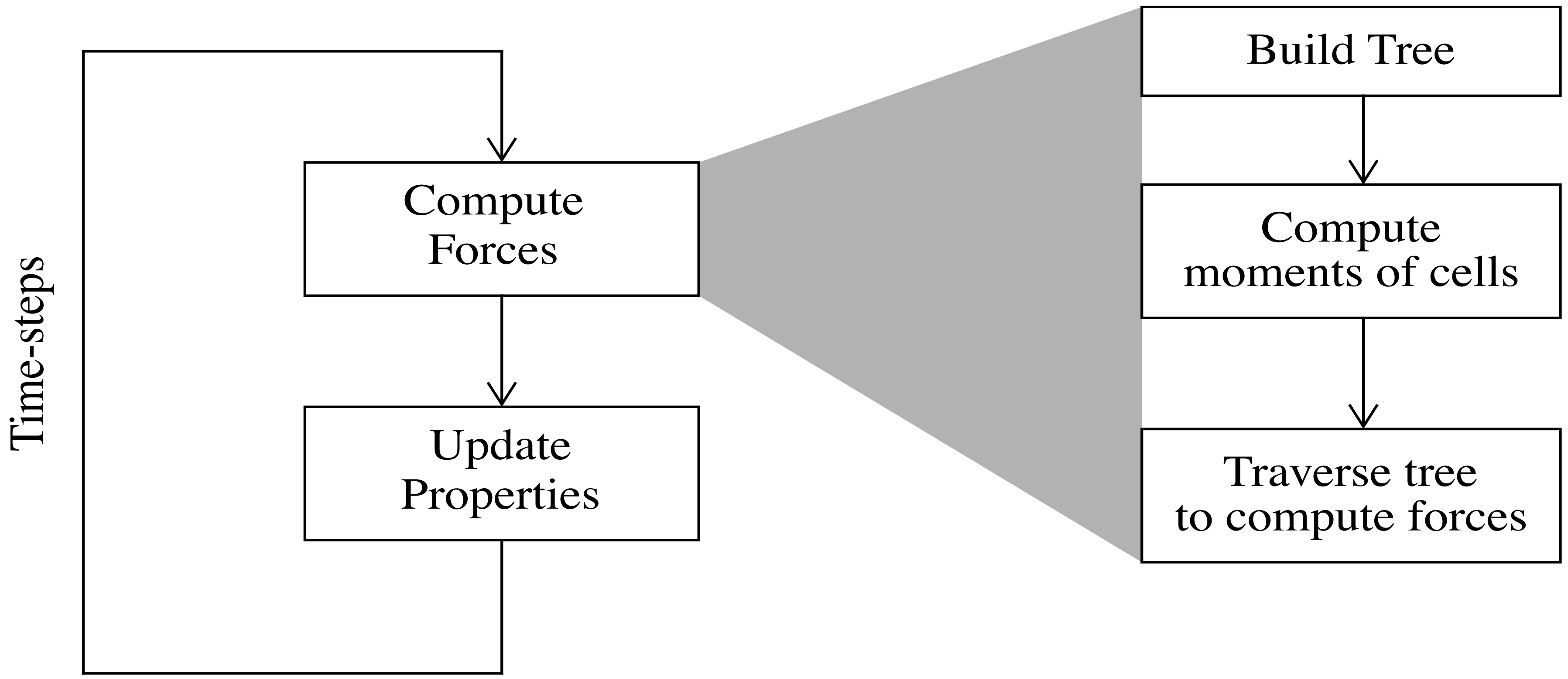
- Performance issues trade off with one another; iterative refinement
- Ready to understand using workloads to evaluate systems issues

- Introduction
- Load Balance and Synchronization Wait Time
- Reducing Inherent Communication
- Reducing Extra Work
- **Applications**

Applications

- Barnes-Hut
- Raytrace
- Ocean

Barnes-Hut: High-level Algorithm



Barnes-Hut

- The galaxy simulation has irregular and dynamically changing behavior
- Time steps compute the net force on every body, and thereby updating positions
- Large group of bodies may be approximated as single body

Barnes-Hut

- Represent the 3D space as quad-tree/oct-tree
  - Internal nodes represent “region of space”
  - Leaves contain individual bodies
- Adaptive
  - Extends high-density region with more levels
- Positions of the bodies change
  - Tree is rebuilt each time step

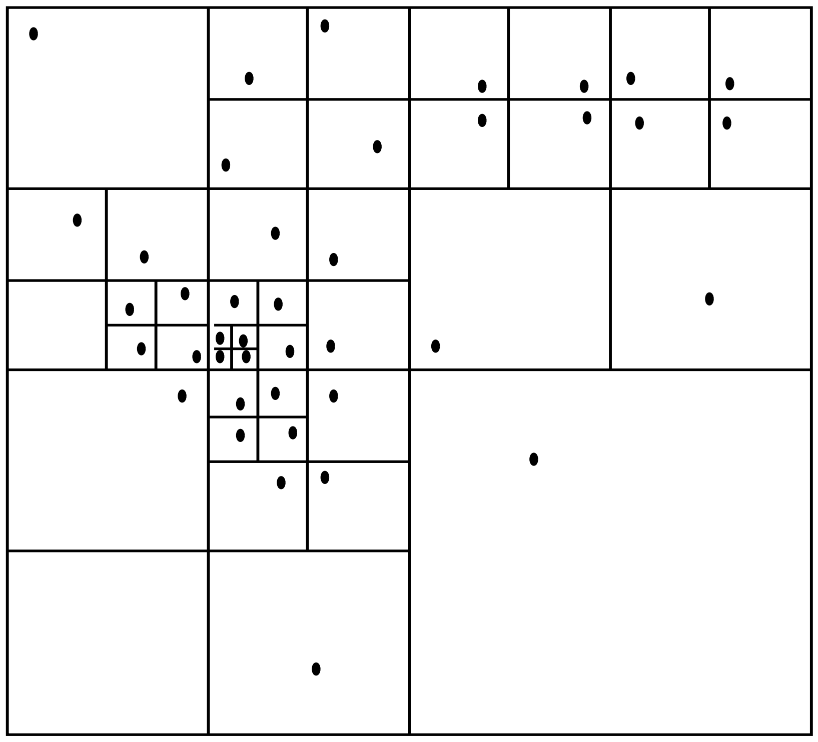
Decomposition and Assignment

- The natural unit of work is a body
  - Except when computing a cell’s center of mass, where unit of work is a cell.
- Amount of work per body and communication is not uniform
  - Bodies keep changing locations
  - Makes load balance difficult

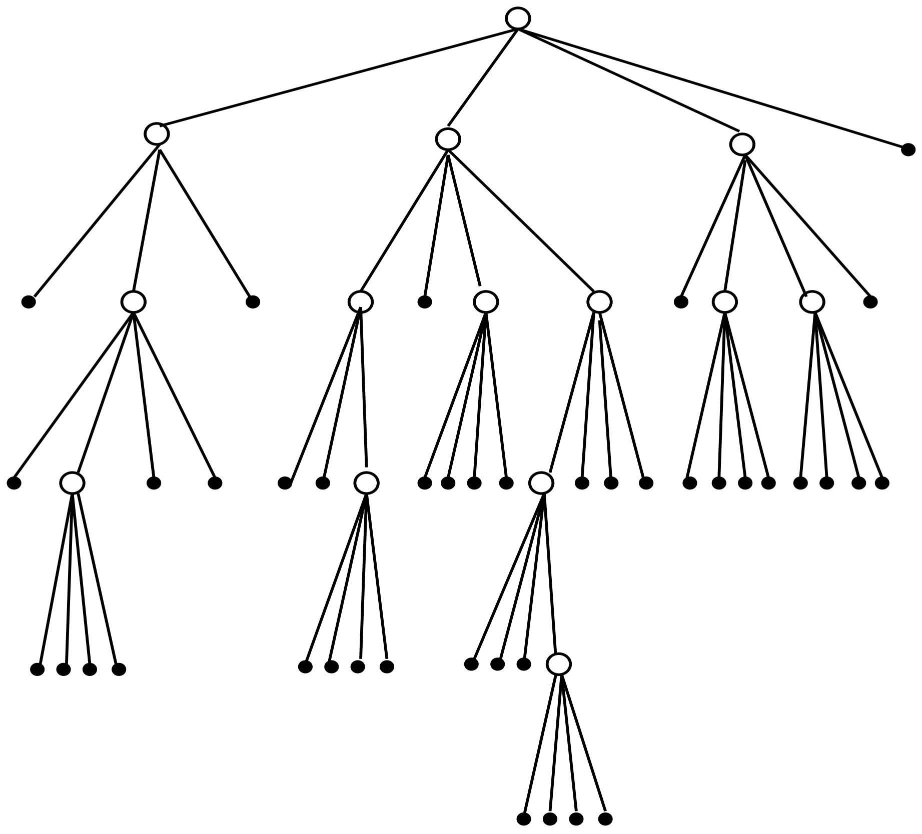
Decomposition and Assignment

- Focus on partitioning on the force calculations
  - Most time consuming step
- Use semi-static partitioning
  - Spatial distribution of bodies changes slowly
- Measure the work done by each particle

Barnes-Hut Data Structure: Quadtree

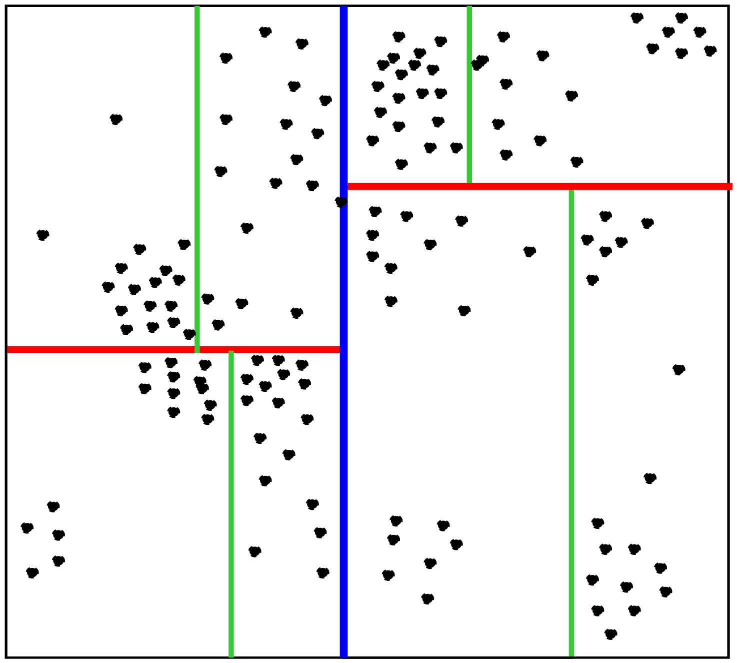


(a) The Spatial Domain

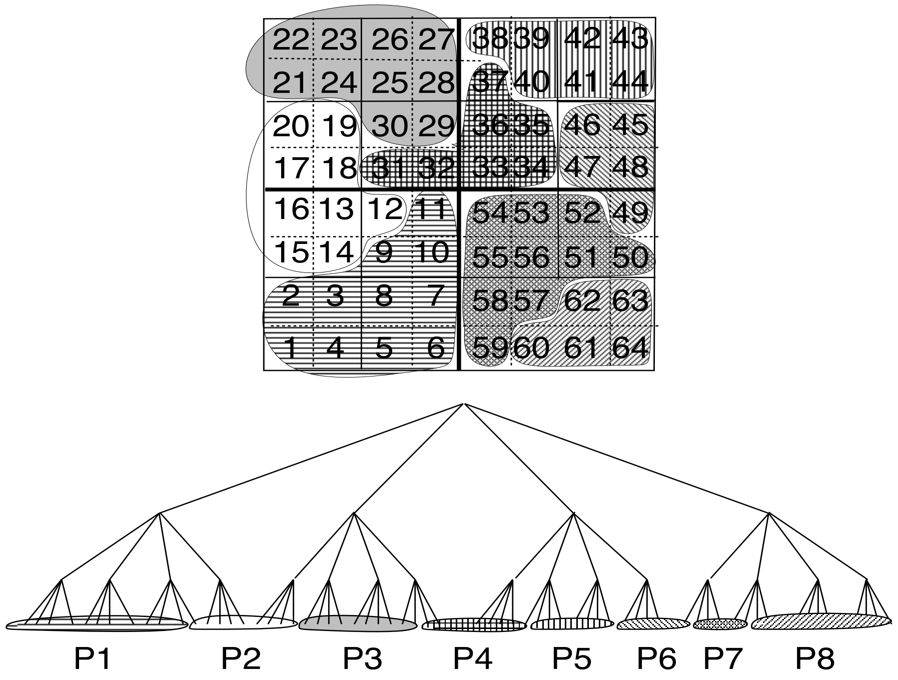


(b) Quadtree Representation

Barnes-Hut: Decomp. and Assignment

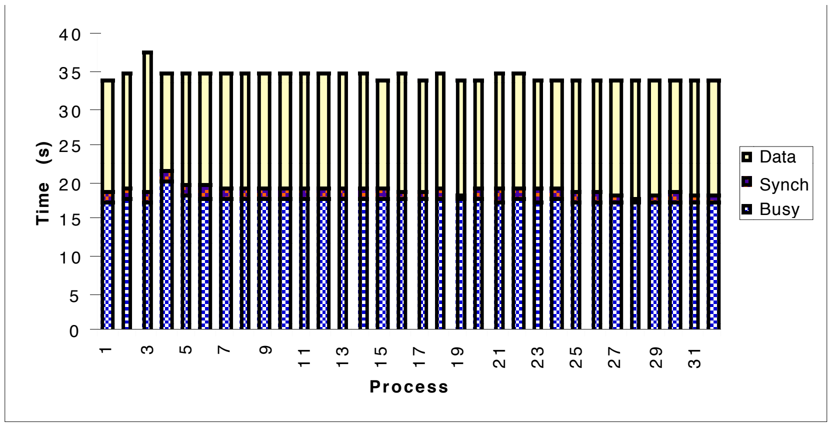


(a) ORB

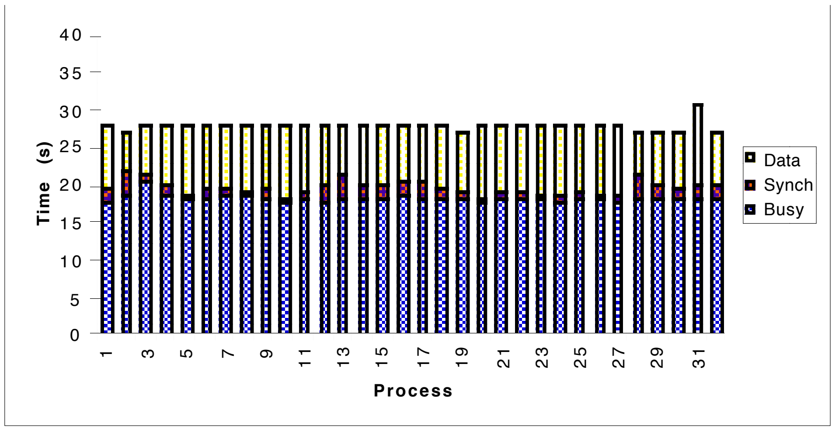


(b) Costzones

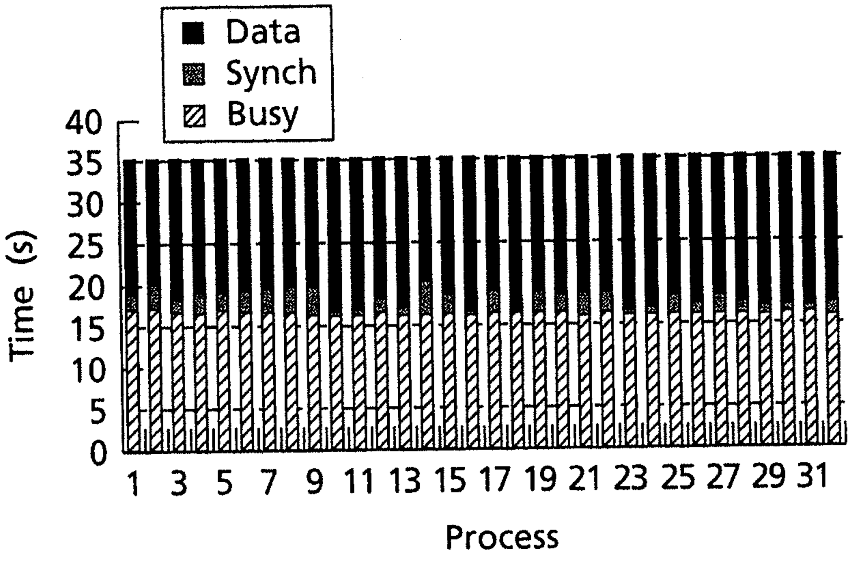
Barnes-Hut: Runtime



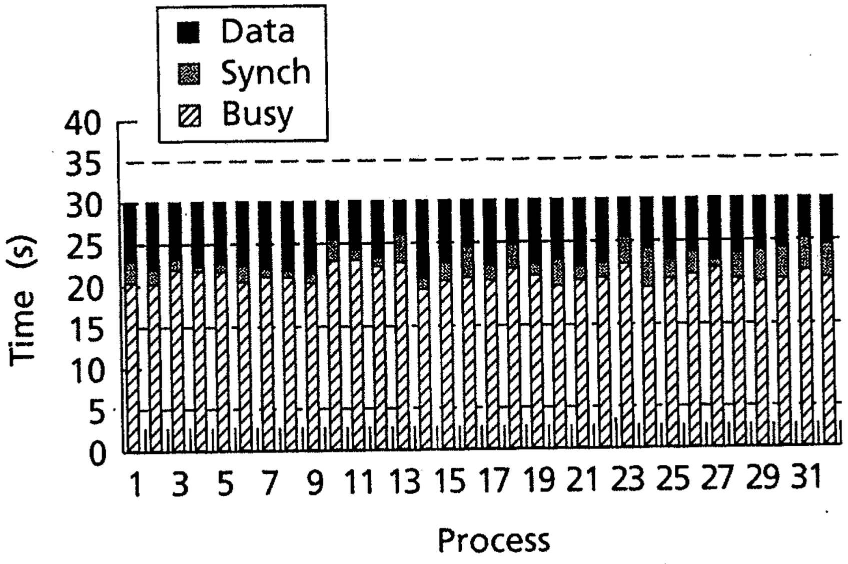
(a) Static assignment of bodies



(a) Semi-static, costzones assignment



(a) Static assignment of bodies

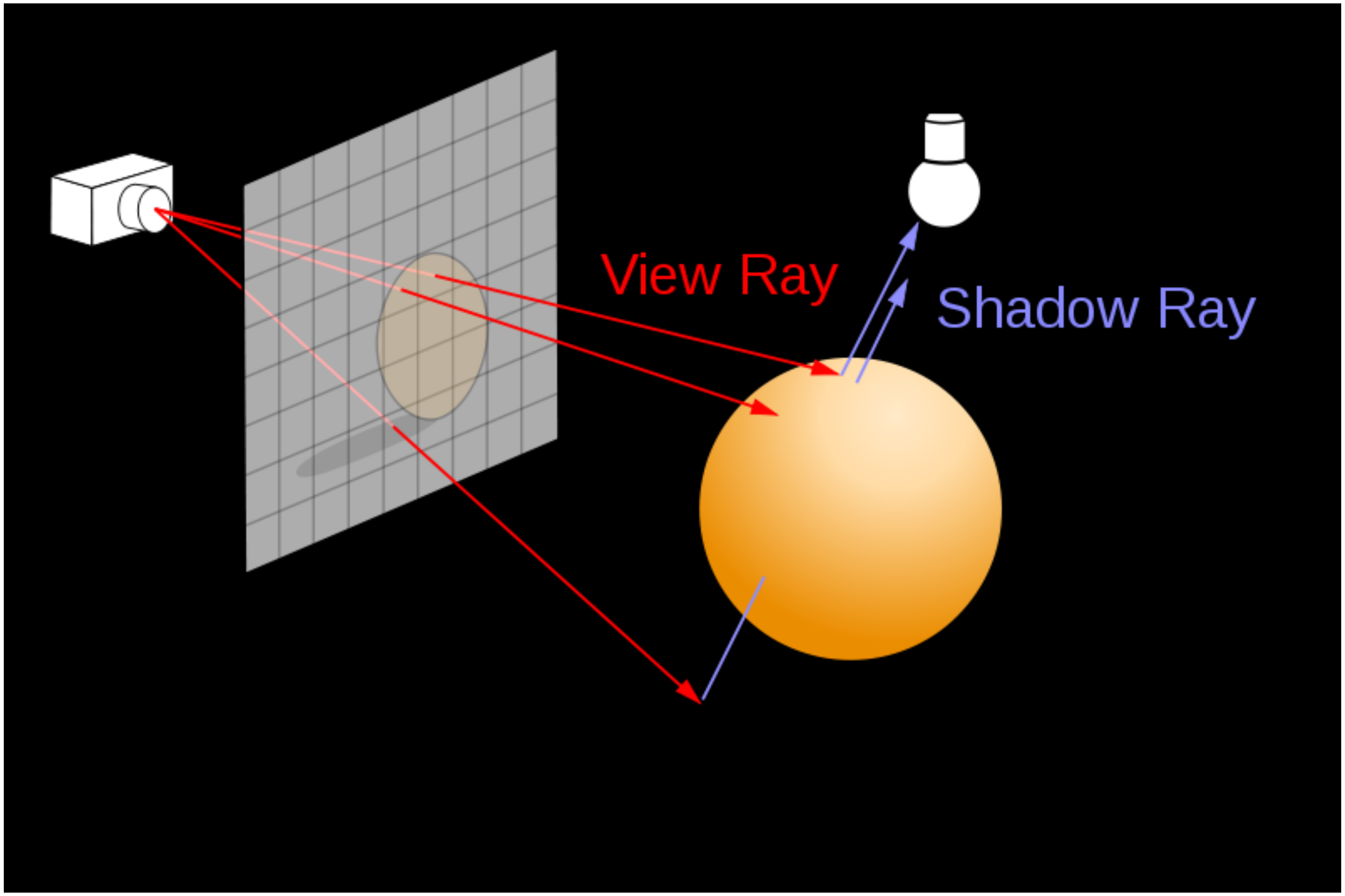


(b) Semistatic costzones assignment

Barnes-Hut Summary

- Irregular, fine-grained, time-varying communication and data access patterns
- Partitioning not obvious
  - must get insight from the application domain
- Load balance quite good with semi-static partitioning

Raytrace



Raytrace

- Rays shot through the pixels in image plane into a 3D scene
  - Path of each ray is traced as it bounces around to determine color / opacity of corresponding pixel
- Hierarchical representation of space called Hierarchal Uniform Grid

Decomposing and Assignment

Two ways:

- Divide the rays
  - Can be finer-grained
  - Different processes may handle rays generated by the same primary ray
  - Easy to program

- Divide the space into subspaces 🡐 use this one
  - Scene-oriented approach
  - Preserves more locality

Decomposing and Assignment

- Rays may encounter different # of reflections
  - Use domain composition to initially assign rays to processes
  - Use dynamic distributed queues with task stealing to balance workload
- No communication except for task stealing
  - 3D scene is read only

Raytrace: Decomp. and Assignment

- Use decentralized queues, dynamic tasking
- Divide volume of 3D space into blocks and tiles

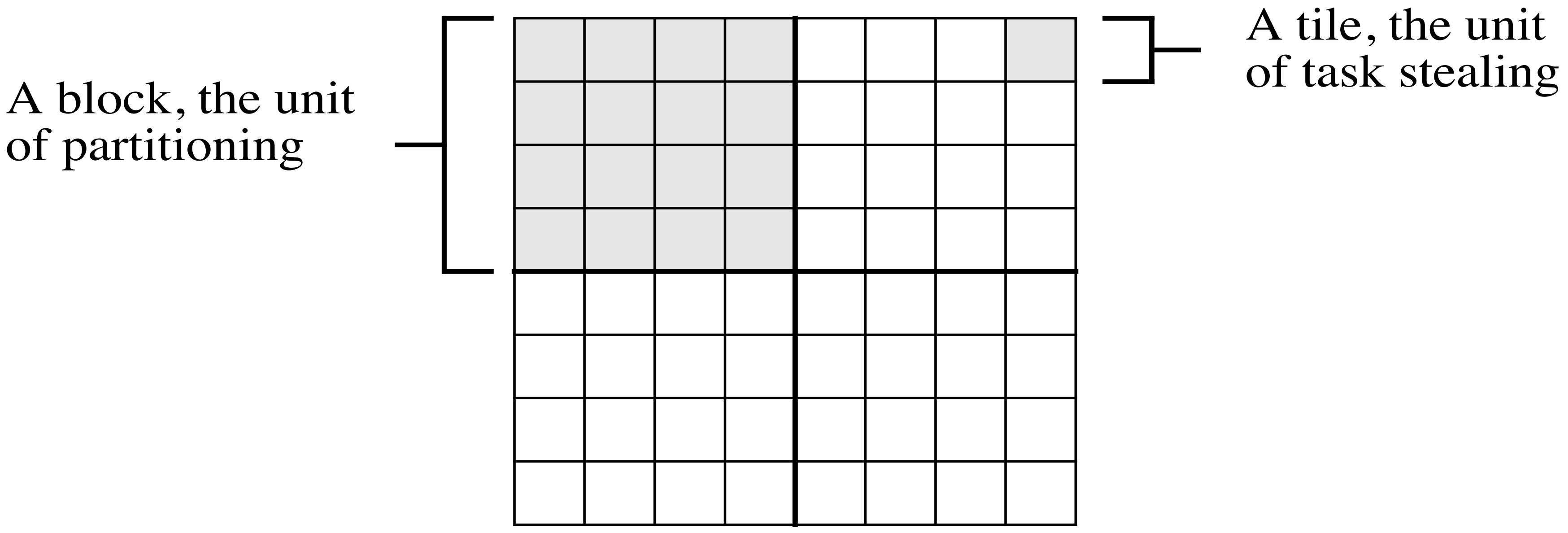

A block, the unit of partitioning
A tile, the unit of task stealing

Figure 3-20 Image plane partitioning in Raytrace for four processors.

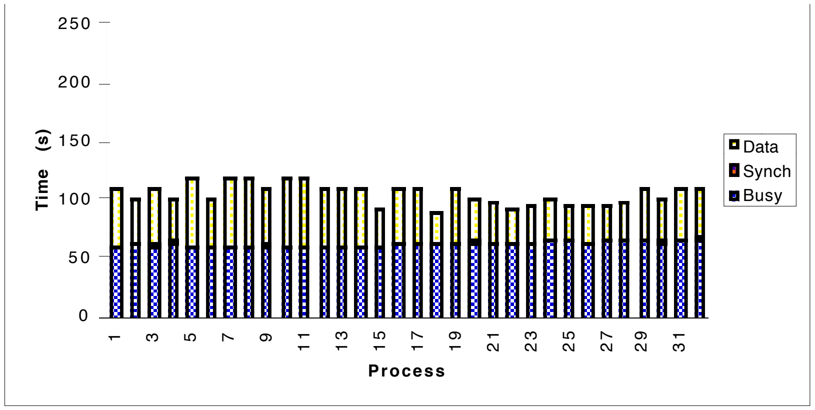
Orchestration

- Spatial Locality
  - The block assignment preserves spatial locality quite well
  - Rays bounce unpredictably, hard to preserve locality for each ray
- Temporal Locality
  - Domain decomposition and spatial coherence increase temporal locality

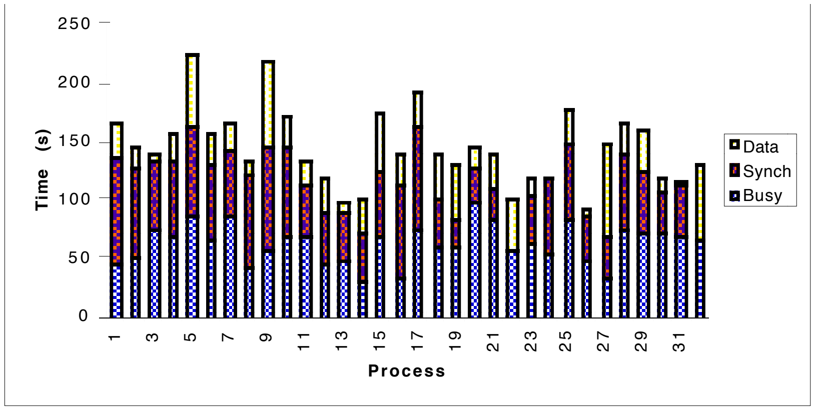
Orchestration

- Synchronization with Locks
  - Used for queues
  - Used for some global variables (statistics)
  - Used at end of rendering

Raytrace: Runtime

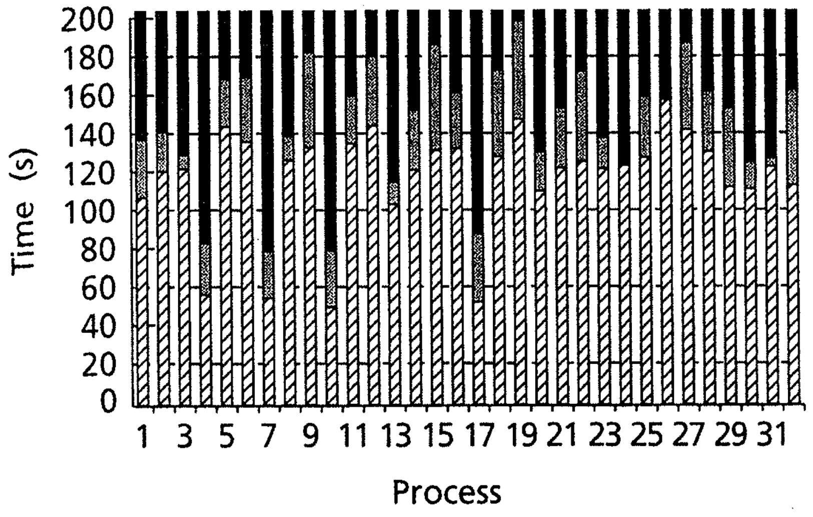


(a) With task stealing

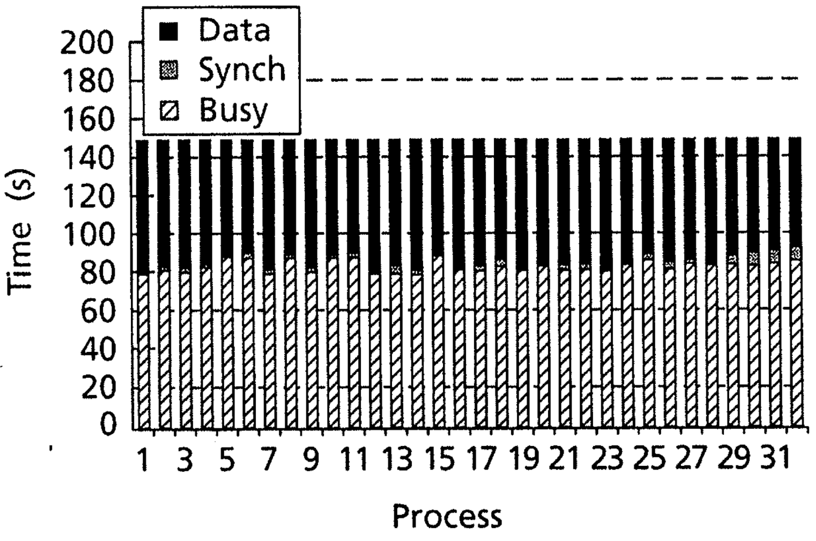


(a) Without task stealing

Figure 3-21 Execution time breakdowns for Raytrace with the balls data set on the Origin2000.



(a) Without task stealing



(b) With task stealing

Ocean

- Simulates currents in an ocean basin
- Important application in fluid dynamics

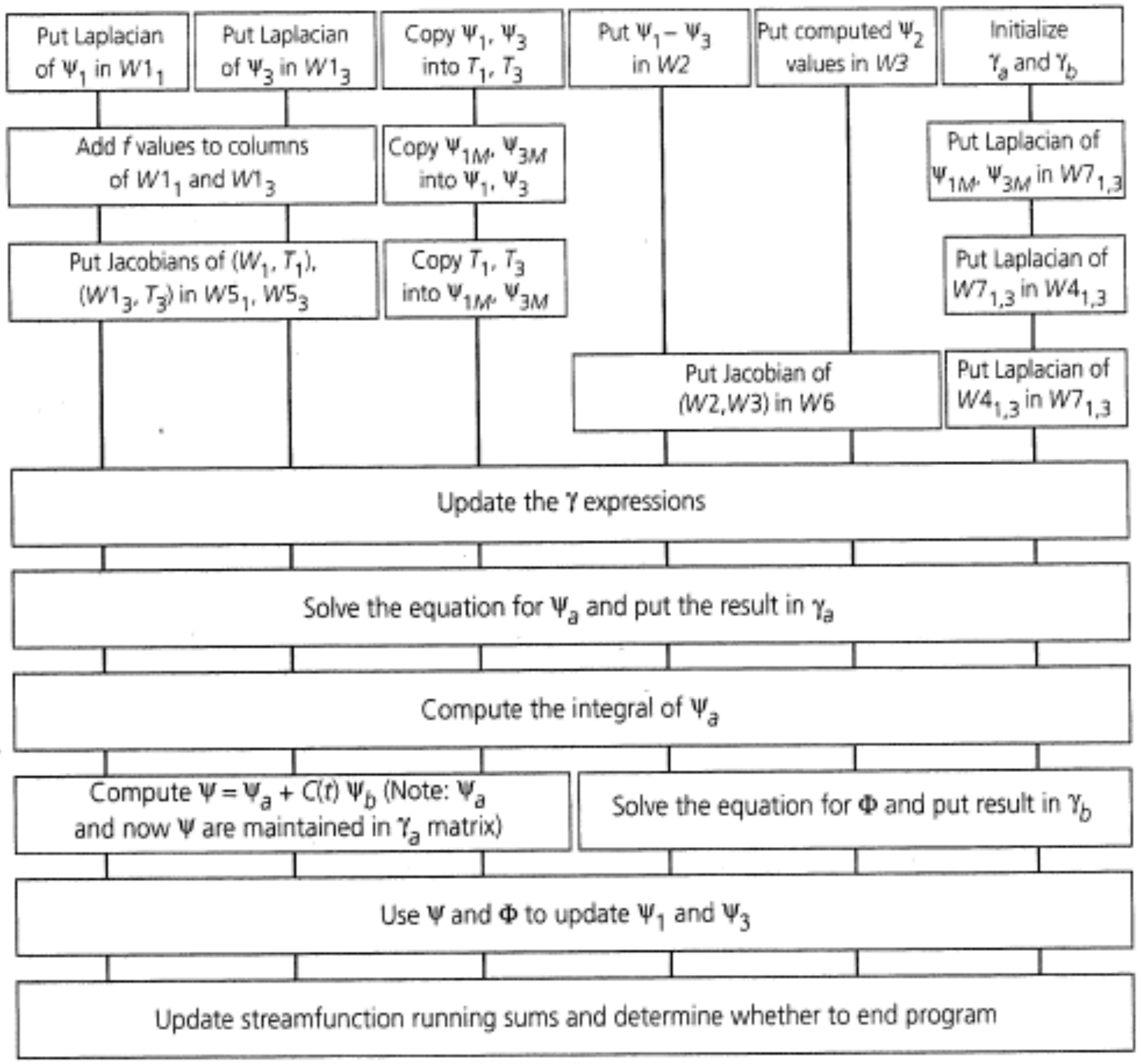

Put Laplacian of Ψ_1 in $W1_1$
Put Laplacian of Ψ_3 in $W1_3$
Copy Ψ_1, Ψ_3 into T_1, T_3
Put $\Psi_1 - \Psi_3$ in $W2$
Put computed Ψ_2 values in $W3$
Initialize γ_a and γ_b
Add f values to columns of $W1_1$ and $W1_3$
Copy Ψ_{1M}, Ψ_{3M} into Ψ_1, Ψ_3
Put Laplacian of Ψ_{1M}, Ψ_{3M} in $W7_{1,3}$
Put 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

FIGURE 3.14 Ocean: The phases in a time-step and the dependences among grid computations. Each box is a grid computation (or pair of similar computations). Computations connected by vertical lines are dependent while others, such as those in the same row, are independent. The parallel program treats each horizontal row as a phase and synchronizes between phases.

Decomposition and Assignment

- Concurrency at two levels
  - Grid computation (functional-level)
  - Within each grid computation (data-level)
- Not enough concurrency across grid computations
  - Work associated with different grid computations is quite varied and depends on problem size

Decomposition and Assignment

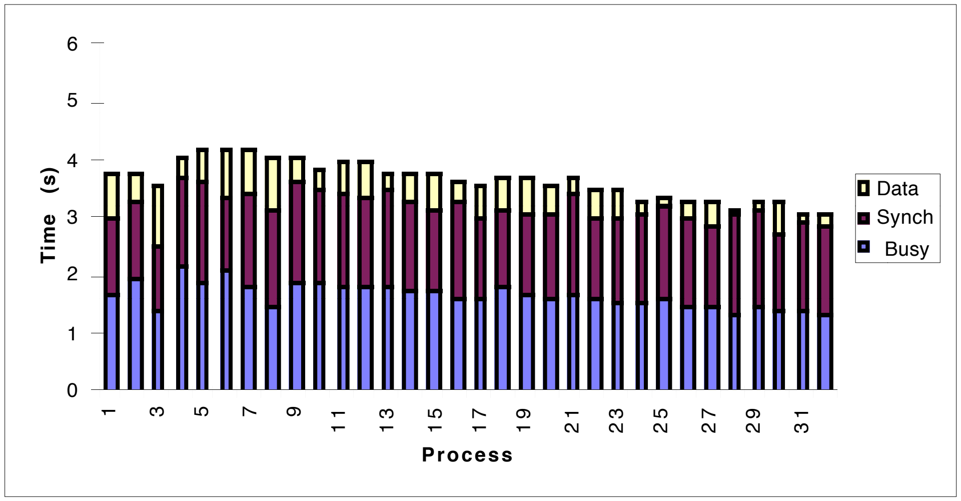
- Each of the grid computations are
  - Fully data parallel
  - Each grid point does roughly the same work
- Use block structure

- Trade off between load balance and data locality in assigning border points

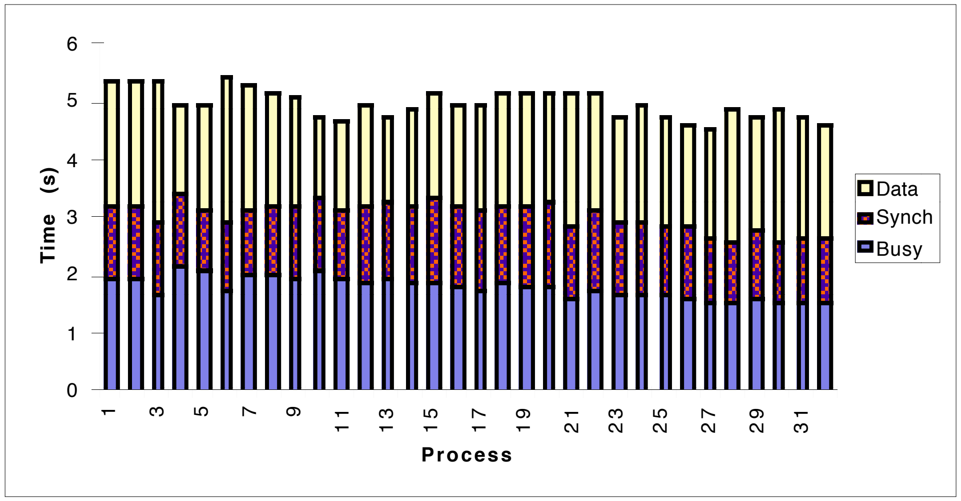
Orchestration and Mapping

- Spatial Locality
  - Very good spatial locality, better in row-oriented partitions
- Synchronization
  - Global barrier to sync between computational phases, as well between the sweeps of multi grid
- Mapping
  - Processes whose partitions are adjacent run on processors near each other

Ocean: Runtime



(a) Four-dimensional arrays

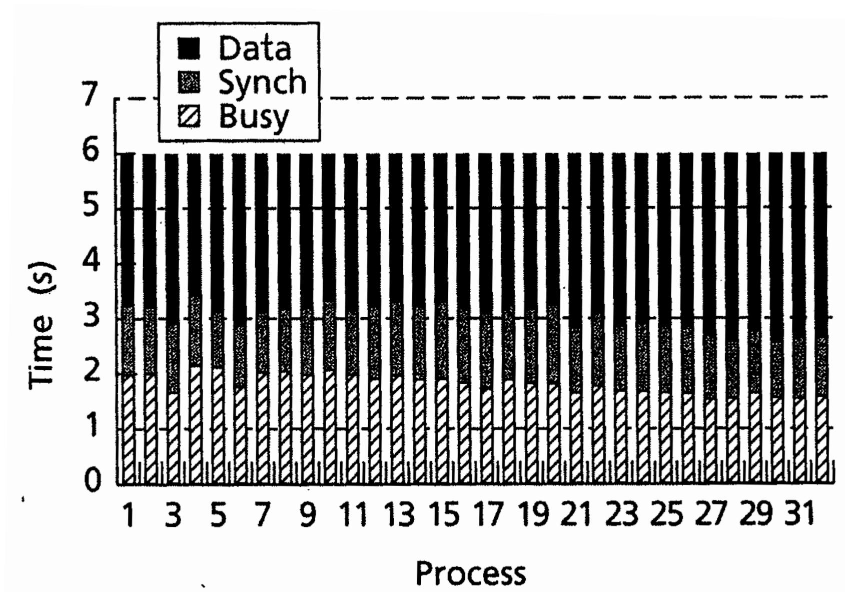


(a) Two-dimensional arrays

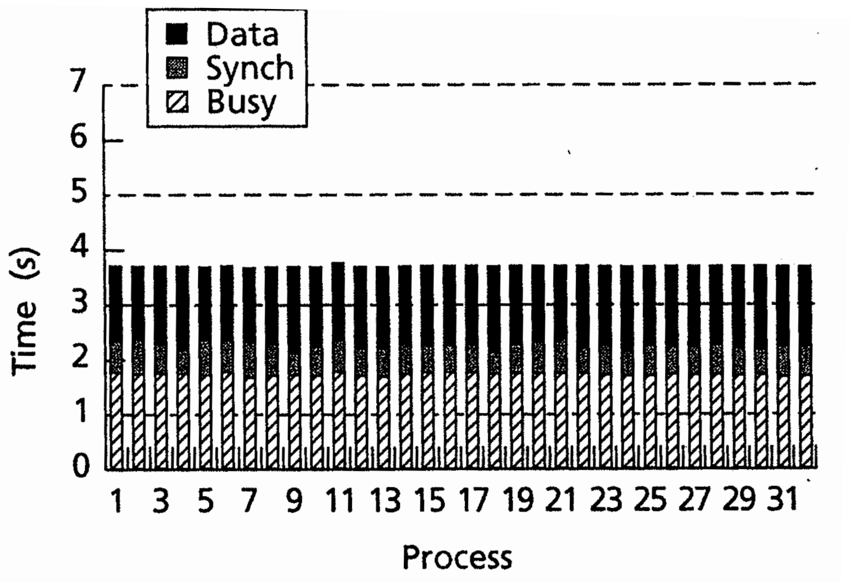
Figure 3-15 Execution time breakdowns for Ocean on a 32-processor Origin2000.

The size of each grid is 1030-by-1030, and the convergence tolerance is 10^{-3}. The use of four-dimensional arrays to represent the two-dimensional arrays to represent the two-dimensional grids clearly reduces the time spent stalled on the memory system (including communication). This data wait time is very small because a processor's partition of the grids it uses at a time fit very comfortably in the large 4MB second-level caches in this machine. With smaller caches, or much bigger grids, the time spent stalled waiting for (local) data would have been much larger.

Ocean: Runtime



(a) Two-dimensional arrays



(b) Four-dimensional arrays

Ocean Summary

- Ocean is a good representative of many applications that stream through regular arrays
- The efficiency increases with grid size