Intro to Parallel Programming

Outline

1. Intro – Why parallel computers?

2. Why is parallel programming ≠ serial programming?

   → Case Study: Try to parallelize serial SOR

3. Intro to Parallel Programming

   → Overall ideas (DAOM): **D**ecomposition / **A**ssignment / **O**rchestration / **M**apping

4. Case Study Revisited → Parallel SOR

Part 1: Why Parallel Computers?

Parallel computers deliver cost-effective performance for certain applications for which this is not possible with uniprocessors (single core).

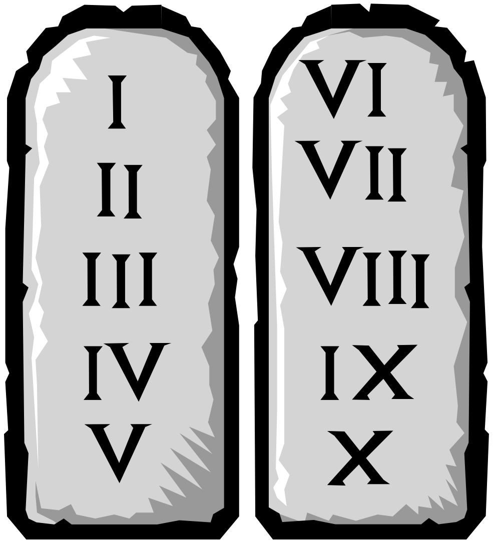
Holy grail of parallel computer architecture:

Be able to put together a very large number of very inexpensive processors and easily get performance proportional to that number of processors.

Why Parallel Computers?

Magic!!

Why parallel computers?



Fundamental Law of Parallel Architecture:

If you have an application that runs in time T on a uniprocessor, then the time it takes to run on a parallel computer with P processors can be as little as T/P.

More basics – But aren't serial processors really fast?

- Why not wait for Moore's law to get you the performance you need?
  - P = 2 ⇔ 18 months, P = 4 ⇔ 3 years, etc.

  → *Aside: Is Moore's law failing?*

- Other virtues of parallel computing
  - Fault tolerance
  - Programmability (e.g. Dennis's argument)

Isn’t this easy?? – NO!

- Typical high-end applications involve modeling physical systems.
- Let application performance be characterized by T = cn^4 where n is the problem size.
  - *n^3 for resolution in 3 spatial dimensions and n for resolution in time*
- Then a factor of P speedup increases the problem size that can be solved by (only) $P^{1/4}$!

Ex: Climate modeling …

Corollary of Modest Potential (from Larry Snyder):

Parallelism offers only a modest potential benefit.

- Recall Fundamental Principle? Restate it:
- *The maximum speed-up obtainable from using P processors is P.*

It gets more challenging …

Look at how processes are created:

Application → Algorithm → Program → Executable → Process

Where can we lose speed-up over the *factor-of-P* case?

1. Breaking application into P equal sized components (decomposition and load balancing)
2. Inherent algorithmic limitations of parallelizing task (dependencies)
3. Coding inefficiencies (beyond serial ones)
4. Compiler inefficiencies (beyond serial ones)
5. Programming model mismatches
6. Communication/synchronization overhead
    - Physical limits of bandwidth/latency
    - System/protection overhead
    - Contention within the communication network
7. Extra work – dynamic actions to improve performance

Part 2: How to Program Parallel Computers?
First attempt: Try Sequential Code

Big Problem: How do we program a parallel computer?

Idea ➔ Simply program as always and let the tools (compiler, etc.) make them run efficiently on parallel computers

Q: Why (might this work)?

A: 40 years of work in parallelizing compilers.

Sketch of approach: Analyze and optimize program. We've already seen →

- Find dependencies ("hazards" in ILP)
- Schedule code in basic blocks
- Loop transformations – interchange, blocking
- Loop unrolling (also - splitting, fusion)
- Strip mining

Use methods outlined in next section, but automate

Try Sequential Code?

Fundamental Problem:

- Uniprocessor languages follow uniprocessor programming model. In particular, *instructions execute sequentially and in program order*
- Issue: ***Even if the operations do not need to be sequential, the programmer will often introduce unnecessary dependencies that force them to be executed sequentially.***

Recall: True (data) dependences occur when one instruction needs data computed by another instruction (RAW). For pipelined and superscalar data paths we were able to schedule instructions after identifying the data dependences. This improved performance by removing stalls. (Loop unrolling helped.) What's different now?

Intro to High Performance Compilers

Job of the compiler ➔ Transform a computation from a high-level representation that is easy for a human to understand into a low-level representation that a machine can execute.

Problem ➔ high-level representation does not accommodate details of architecture (for good reasons). Naïve translation is likely to introduce inefficiencies into machine code.

Goal of Optimization ➔ Eliminate these inefficiencies.

Procedure ➔

- Program is a specification not a recipe. Can be transformed as long as it executes correctly (for all inputs, outputs identical to unoptimized)
- Fundamental challenge of optimization: How can we tell that the program is still correct? E.g., parallelizing code changes order that instructions execute.
- Methods involve finding dependences: must preserve order of loads and stores to each location.
    - Well, not exactly, but it's a good start, and you won't go wrong.

Scalar Example

Let <Si,Sj,…,Sz> denote a dependence relation among instructions. That is, Si must execute before Sj in any valid reordering of the program.

```
S1      PI = 3.14159;
S2      R = 5;
S3      AREA = PI * R ^ 2;
```

If execution order must be preserved, then <S1,S2,S3>
But clearly <S2,S1,S3> is also correct.
So <S1,S3> and <S2,S3> are dependences but <S1,S2> is not.

Vector Example 1

```
VLOAD  V1,A          // Vector assembly language
VLOAD  V2,B
VADD   V3,V1,V2
VSTORE C,V3
```

Note: The semantics of this code is plain SIMD. We assume that each instruction operates on a vector of data (say, 64 elements) and that each instruction completes before the next begins. At least, must appear as if it does, even if there are HW opts such as chaining.

```
// Fortran 90 array assignment syntax
C(1:64) = A(1:64) + B(1:64)
```

This syntax assumes the same semantics as the V.A.L. above.

```
// But most Fortran is written like this, even for vector code
DO I = 1, 64
   C(I) = A(I) + B(I)
ENDDO
```

No problem: the ordering change does not change the outcome, so this too can be translated into the same V.A.L. This code is ***vectorizable.***

Vector Example 2

```
// What about --
DO I = 1, 64
   A(I+1) = A(I) + B(I)
ENDDO
```

Note: each iteration uses a result from the previous iteration.

```
// Not the same as above!
A(2:65) = A(1:64) + B(1:64)
```

Note: all inputs refer to the original array values. This code is ***not vectorizable***

```
// What about this?
DO I = 2, 65
   A(I-1) = A(I) + B(I)
ENDDO
```

```
// IS the same as above …
A(1:64) = A(2:65) + B(2:65)
```

This code ***is vectorizable!***

Parallelize through data decomposition

Principal strategy ➔ look for data decomposition in which parallel tasks perform similar operations on different elements of the data arrays.

Method ➔ given an input program that consists of nests of loops, use a constraint system that preserves the order of data accesses.

Bernstein's conditions ➔ Iterations I1 and I2 can be safely executed in parallel if

1. iteration I1 does not write into a location that is read by iteration I2
2. iteration I2 does not write into a location that is read by iteration I1
3. iteration I1 does not write into a location that is also written by iteration I2

```
// Parallelizable?
DO I = 1, N
   A(I) = A(I) + B(I)
ENDDO
```

```
// Parallelizable?
DO I = 1, N
   A(I+1) = A(I) + B(I)
ENDDO
```

```
// Parallelizable?
DO I = 1, N
   S = A(I) + B(I)
ENDDO
```

```
// Parallelizable?
DO I = 1, N
   A(I-1) = A(I) + B(I)
ENDDO
```

Sometimes we can relax (a little)

```
// Vectorizable?
DO I = 1, N
   T = A(I)
   A(I) = B(I)
   B(I) = T
ENDDO
```

```
// Vectorizable?
DO I = 1, N
   T(I) = A(I)
   A(I) = B(I)
   B(I) = T(I)
ENDDO
```

Valid Transformation ≡ A transformation is valid for the program to which it is applied if it preserves all dependences in the program.

```
// Can we interchange L1 with {S1,S2,S3}?
L0      DO I = 1,N
L1         DO J = 1,2
S0            A(I,J) = A(I,J) + B
           ENDDO
S1         T = A(I,1)
S2         A(I,1) = A(I,2)
S3         A(I,2) = T
        ENDDO
```

Valid Transformations preserve a condition that is stronger than equivalence

Example: SOR

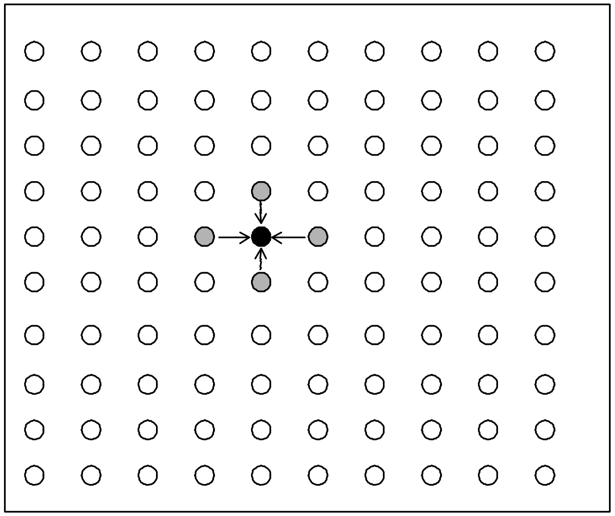
SOR ⇔ Successive Over Relaxation.

- *Simple stencil codes are key in many apparently complex applications*

Relaxation-based algorithm: each non-boundary value gets weighted average of neighbors until steady state is reached

Example: Soap bubble (film) in a wire frame.

10	9	8	7
9	?	?	6
8	?	?	5
7	6	5	4



Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] + A[i,j + 1] + A[i + 1, j])

Serial Code

Note reuse of array storage. Saves some storage, but serializes the computation by introducing a(n) (perhaps) unneeded dependency.

```
while (not converged)
  for i = 1 to n-1 do
    for j = 1 to n-1 do
      A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[j,j+1] + A[i+1,j])/5
```

10	9	8	7
9	0	0	6
8	0	0	5
7	6	5	4

10	9	8	7
9	3.6	3.5	6
8	3.5	3.4	5
7	6	5	4

10	9	8	7
9	5.7	5.3	6
8	5.3	4.8	5
7	6	5	4

10	9	8	7
9	6.9	6.2	6
8	6.4	5.5	5
7	6	5	4

10	9	8	7
9	7.5	6.6	6
8	6.7	5.8	5
7	6	5	4

……

10	9	8	7
9	8	7	6
8	7	6	5
7	6	5	4

Assume:

- **I** iterations for convergence
- **NxN** array

complexity = $O(IN^2)$

```
int n;                                  /*size of matrix: (n + 2-by-n + 2) elements*/
float **A, diff = 0;

main()
begin
   read(n) ;                            /*read input parameter: matrix size*/
   A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
   initialize(A);                       /*initialize the matrix A somehow*/
   Solve (A);                           /*call the routine to solve equation*/
end main

procedure Solve (A)                     /*solve the equation system*/
   float **A;                           /*A is an (n + 2)-by-(n + 2) array*/
begin
   int i, j, done = 0;
   float diff = 0, temp;
   while (!done) do                     /*outermost loop over sweeps*/
      diff = 0;                         /*initialize maximum difference to 0*/
      for i ← 1 to n do                 /*sweep over nonborder points of grid*/
         for j ← 1 to n do
            temp = A[i,j];              /*save old value of element*/
            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
               A[i,j+1] + A[i+1,j]);    /*compute average*/
            diff += abs(A[i,j] - temp);
         end for
      end for
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

Decomposition and Mapping

- Idea -- whatever is independent can be run concurrently.
- Try to "decompose" the computation into those independent pieces.

- Simple way to identify concurrency is to look at ***loop iterations***
  - dependence analysis; if not enough concurrency, then look further
  - Not much concurrency here at this level (all loops sequential)

```
L0     while (not converged)
L1       for i = 1 to n do
L2         for j = 1 to n do
             A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                       A[j,j+1] + A[i+1,j])/5
```

Still, let's try to map

- each inner loop iteration to its own distinct process
- each process to its own distinct processor

# of elements = # of processors = N^2

Look at Dependences

```
L0  while (not converged)
L1    for i = 1 to n do
L2      for j = 1 to n do
          A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j])/5
```

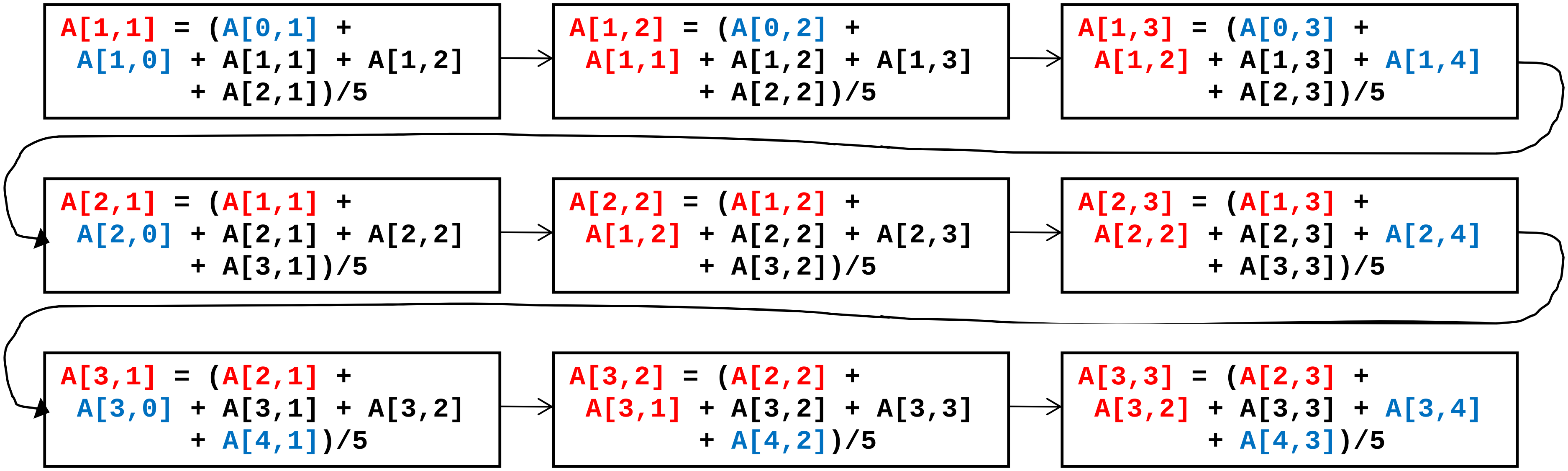
Execution order (i,j): *→ (how) can these be reordered?*

RED – values computed on **THIS** iteration (K+1)

BLUE – **Constants**

BLACK – values computed on **PREVIOUS** iteration (K)

```
A[i,j] = (A[i,j] + A[i,j-1]
         + A[i-1,j] + A[i,j+1]
         + A[i+1,j])/5
```



```
A[1,1] = (A[0,1] +
 A[1,0] + A[1,1] + A[1,2]
          + A[2,1])/5
```

```
A[1,2] = (A[0,2] +
 A[1,1] + A[1,2] + A[1,3]
          + A[2,2])/5
```

```
A[1,3] = (A[0,3] +
 A[1,2] + A[1,3] + A[1,4]
          + A[2,3])/5
```

```
A[2,1] = (A[1,1] +
 A[2,0] + A[2,1] + A[2,2]
          + A[3,1])/5
```

```
A[2,2] = (A[1,2] +
 A[1,2] + A[2,2] + A[2,3]
          + A[3,2])/5
```

```
A[2,3] = (A[1,3] +
 A[2,2] + A[2,3] + A[2,4]
          + A[3,3])/5
```

```
A[3,1] = (A[2,1] +
 A[3,0] + A[3,1] + A[3,2]
          + A[4,1])/5
```

```
A[3,2] = (A[2,2] +
 A[3,1] + A[3,2] + A[3,3]
          + A[4,2])/5
```

```
A[3,3] = (A[2,3] +
 A[3,2] + A[3,3] + A[3,4]
          + A[4,3])/5
```

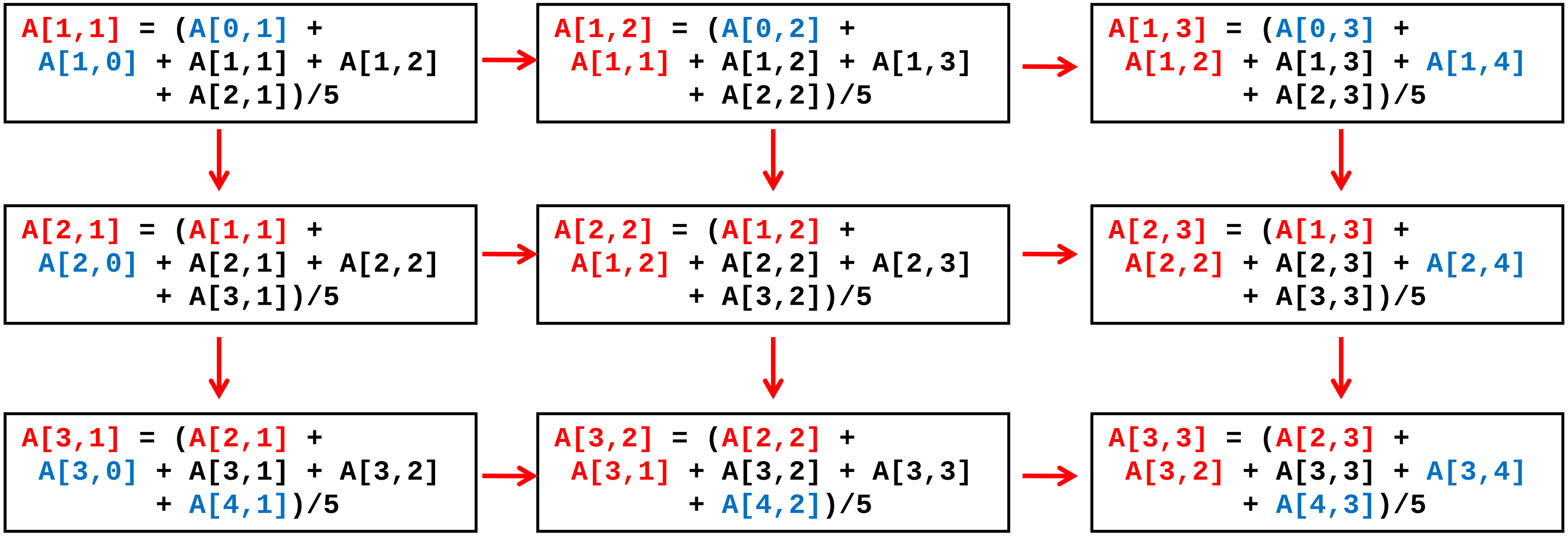
Look at Dependences

```
while (not converged)
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[i,j+1] + A[i+1,j])/5
```

RED – values computed on **THIS** iteration (K+1)

BLUE – **Constants**

BLACK – values computed on **PREVIOUS** iteration (K)



```
A[1,1] = (A[0,1] +
 A[1,0] + A[1,1] + A[1,2]
        + A[2,1])/5
```

```
A[1,2] = (A[0,2] +
 A[1,1] + A[1,2] + A[1,3]
        + A[2,2])/5
```

```
A[1,3] = (A[0,3] +
 A[1,2] + A[1,3] + A[1,4]
        + A[2,3])/5
```

```
A[2,1] = (A[1,1] +
 A[2,0] + A[2,1] + A[2,2]
        + A[3,1])/5
```

```
A[2,2] = (A[1,2] +
 A[1,2] + A[2,2] + A[2,3]
        + A[3,2])/5
```

```
A[2,3] = (A[1,3] +
 A[2,2] + A[2,3] + A[2,4]
        + A[3,3])/5
```

```
A[3,1] = (A[2,1] +
 A[3,0] + A[3,1] + A[3,2]
        + A[4,1])/5
```

```
A[3,2] = (A[2,2] +
 A[3,1] + A[3,2] + A[3,3]
        + A[4,2])/5
```

```
A[3,3] = (A[2,3] +
 A[3,2] + A[3,3] + A[3,4]
        + A[4,3])/5
```

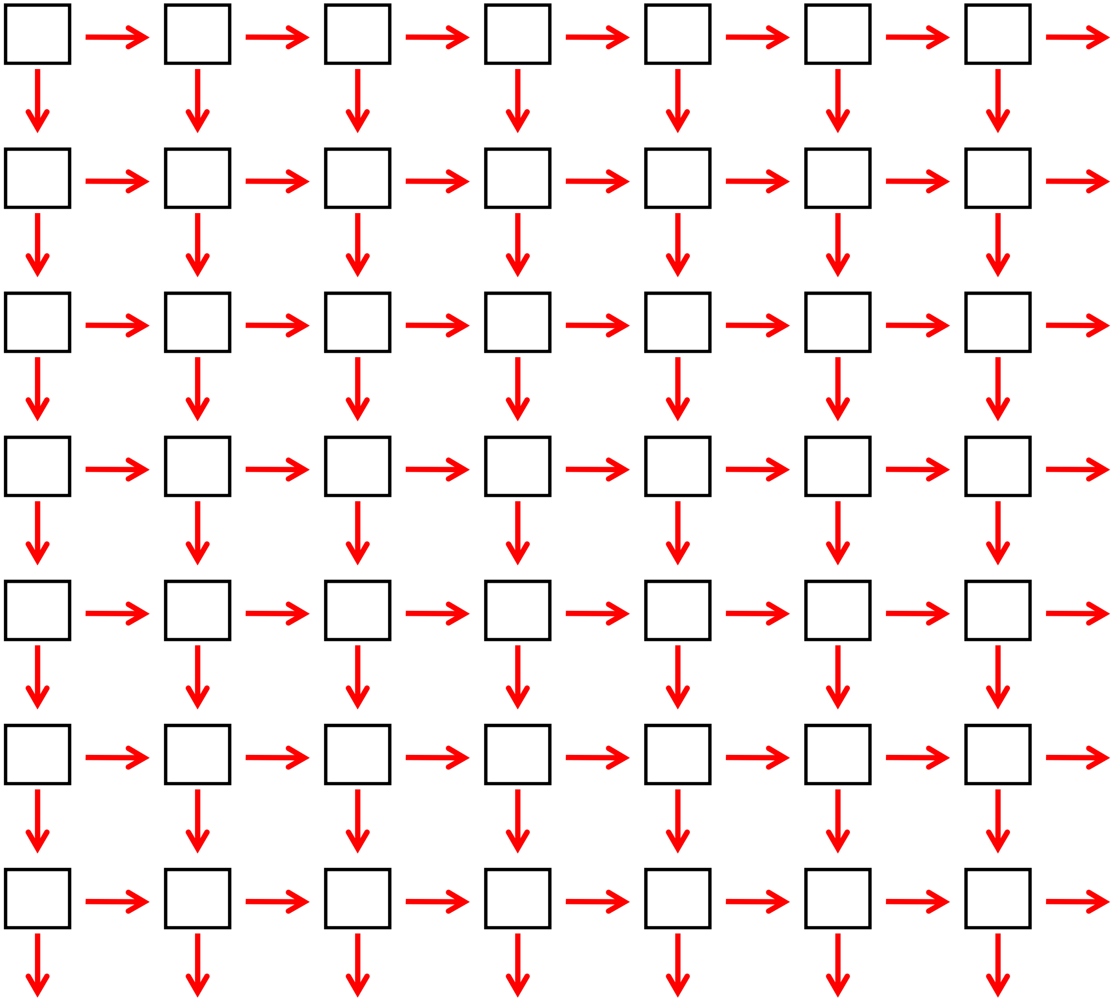
- *Remove execution order*
- ***Arrows show dependences among statements within L0***
- ***A statement can execute when all of the inputs have completed***

Derive Best Execution Flow

```
L0  while (not converged)
L1    for i = 1 to n do
L2      for j = 1 to n do
          A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j])/5
```

To start (for simplicity) →

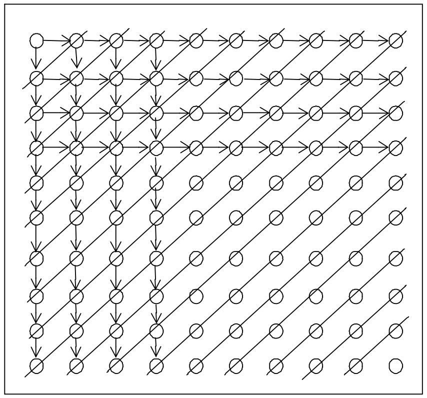
- Assume all PEs execute exactly one inner loop instruction for every iteration of K (L0). And that is instruction (i,j) for PE [i,j].
- Assume that all PEs start iteration K of L0 at the same time.
- PE [i,j] must wait until PEs [i-1,j] and [i,j-1] have executed their instructions.
- Note which PEs can execute in parallel.
- What is the utilization?



NOW → Remove L0 sync.

- As before PE [i,j] must wait until PEs [i-1,j] and [i,j-1] have executed their instructions.
- But this time, PE[1,1] does not need to wait for new iteration of L0 (although it does have to wait for its data to be consumed).
- Note which PEs can execute in parallel.
- What is the utilization?

Parallelized Serial Code (pseudo)



```
// Original serial code
while (not converged)
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[j,j+1] + A[i+1,j])/5
```

// Concurrency *O(n)* along antidiagonals, serialization *O(n)* along diagonal

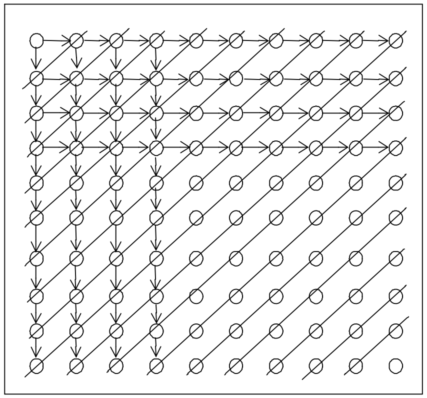
```
// Note – synchronization is non-trial
while (not converged) // One L0 at a time or add another test
  ForEach point  // in parallel
    if (A_ready[i-1][j] == TRUE && A_ready[i][j-1] == TRUE)
      A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[j,j+1] + A[i+1,j])/5
      A_ready[i,j] = TRUE
```

Issues: ***work distribution, load imbalance, sync granularity, task granularity***

Map to an NxN parallel processor

Start Up
Complexity = O(N)

3	(3)	2	(2)	1
(3)	2	(2)	1	(1)
2	(2)	1	(1	0
(2)	1	(1)	0	0
1	(1)	0	0	0
		(a)		



Steady State
Complexity = O(I)

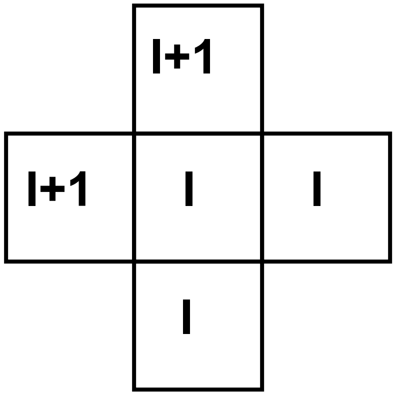
$k+2$	$(k+2)$	$k+1$	$(k+1)$	k
$(k+2)$	$k+1$	$(k+1)$	k	(k)
$k+1$	$(k+1)$	k	(k)	$k-1$
$(k+1)$	k	(k)	$k-1$	$(k-1)$
k	(k)	$k-1$	$(k-1)$	$k-2$

Tear Down
Complexity = O(N)

m	m	m	(m)	$m-1$
m	m	(m)	$m-1$	$(m-1)$
m	(m)	$m-1$	$(m-1)$	$m-2$
(m)	$m-1$	$(m-1)$	$m-2$	$(m-2)$
$m-1$	$(m-1)$	$m-2$	$(m-2)$	$m-3$

Serial code introduces unnecessary dependencies

Each array element computes new value from neighboring values in the iteration shown:



Parallelizer necessarily thinks that the elements up and left must always be computed BEFORE center I.

Therefore: 2n-1 startup phase
2/iterations steady state phase
2n-1 shutdown phase
Complexity of entire computation `= O(n) + O(I)`

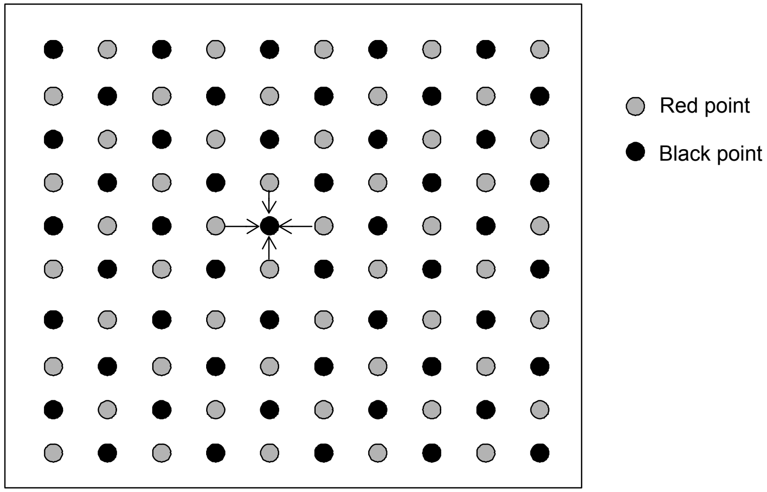
A better-way: Red-Black decomposition

Partition array like a checker board

Red squares compute on even iteration

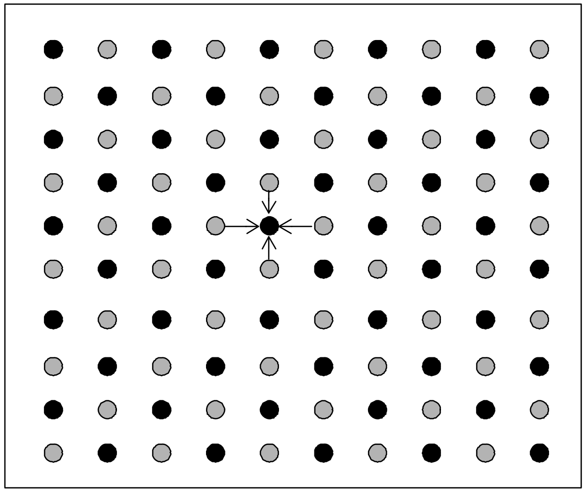
Black squares compute on odd iteration

Therefore 2 cycles per iteration, but entire computation is O(Ic) and independent of n.



Exploit Application Knowledge

Reorder grid traversal: red-black ordering



○ Red point

● Black point

```
while (not converged)
  for i = 1 to n-2 do       // BLACK
    if (i % 2 == 0) jj = 2; // even row
    else jj = 1;            // odd row
    for j = jj to n-1 by 2 do
      A[i,j] = .25 * (A[i,j-1] + A[i-1,j] +
               A[j,j+1] + A[i+1,j])
  for i = 1 to n-2 do        // RED
    if (i % 2 == 0) jj = 1;  // even row
    else jj = 2;             // odd row
    for j = jj to n-1 by 2 do
      A[i,j] = .25 * (A[i,j-1] + A[i-1,j] +
               A[j,j+1] + A[i+1,j])
```

- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)

Red-black dependencies

k+1	*k*	*k+1*	*k*	*k+1*
k	*k+1*	*k*	*k+1*	*k*
k+1	*k*	*k+1*	*k*	*k+1*
k	*k+1*	*k*	*k+1*	*k*
k+1	*k*	*k+1*	*k*	*k+1*

No startup phase
I* iterations steady state phase
No shutdown phase
Complexity of entire computation = `O(I*)`

10	9	8	7
9	0	0	6
8	0	0	5
7	6	5	4

10	9	8	7
9	4.5	0	6
8	0	2.5	5
7	6	5	4

10	9	8	7
9	4.5	5.25	6
8	5.25	2.5	5
7	6	5	4

10	9	8	7
9	7.125	5.25	6
8	5.25	5.125	5
7	6	5	4

10	9	8	7
9	7.125	6.56	6
8	6.56	5.125	5
7	6	5	4

10	9	8	7
9	7.78	6.56	6
8	6.56	5.78	5
7	6	5	4

10	9	8	7
9	7.95	6.89	6
8	6.89	5.95	5
7	6	5	4

......

Moral

Cannot expect dependency analysis to succeed in all cases.

Therefore: many parallelizable applications do not have existing easily parallelizable serial code.

Therefore: serial programming model is not a good candidate for parallel programming model.

Aside: If it were…. Then that would be because compilers would be capable of generating efficient parallel code. Which would mean that, by our definition, serial and parallel programming models are the same.

Part 3: Intro to Parallel Programming

Types of parallelism (mostly loose definitions)

- **Bit-level:** Bigger ALU can process more bits in parallel
- **Instruction Level Parallelism (ILP):** Parallelism that can be exploited by a pipeline or superscalar processor
- **Data Parallel:** Decompose execution of similar independent computations
- **Task Parallel:** Processors execute completely different tasks, or same tasks on distinctly different data

Static versus dynamic versus semi-static task allocation:

- Static → Processes know what they are supposed to work on (e.g., third N/P iterations of a loop). Or by function (PE1 does the filtering, PE2 the reconstruction, PE3 …).
- Dynamic → Allocated dynamically. Often by grabbing work from a queue.

Parallelizing Computation versus Data

- View of much of the discussion is centered around ***computation***
  - Computation is decomposed and assigned (partitioned)
- Partitioning ***data*** is often a natural view too …
  - Computation follows data: *owner computes*
  - Grid example; data mining; High Performance Fortran (HPF)
    - “map” phase of map-reduce
- … but not always:
  - Distinction between comp. and data stronger in many applications

Managing Concurrency

Static versus *Dynamic* techniques

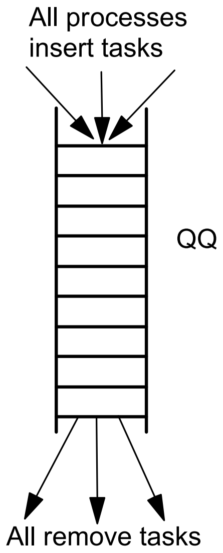
- Static:
  - Algorithmic assignment based on input; won't change
  - Low runtime overhead
  - Computation must be predictable
  - Preferable when applicable (except in multiprogrammed/heterogeneous environment)
- Dynamic:
  - Adapt at runtime to balance load
  - Can increase communication and reduce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

Dynamic Assignment

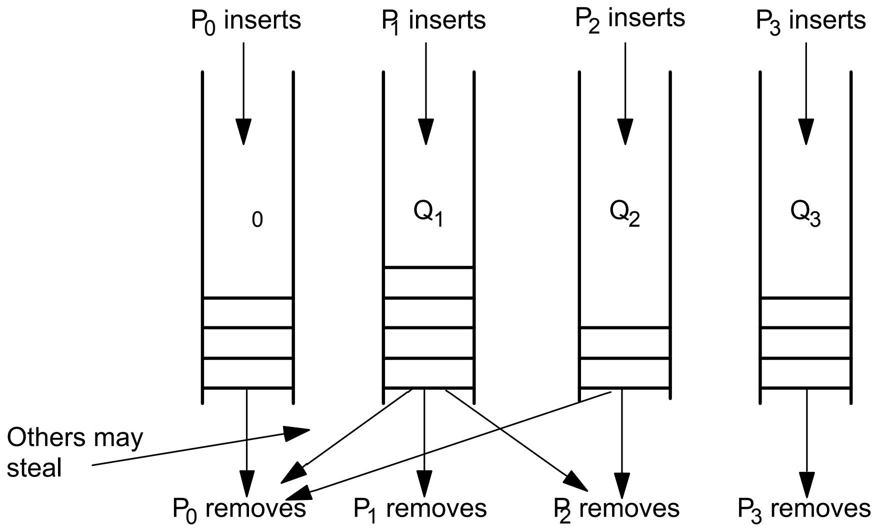
- Profile-based (semi-static):
  - Profile work distribution at runtime, and repartition dynamically
  - Applicable in many computations, e.g. Barnes-Hut, some graphics
- Dynamic Tasking:
  - Deal with unpredictability in program or environment (e.g. Raytrace)
    - computation, communication, and memory 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 pr ocess)

Some preliminaries

Assumption: Sequential *algorithm* is given

- Sometimes need very different algorithm, but beyond scope

Pieces of the job:

1. Identify work that can be done in parallel
2. Partition work, and perhaps data, among processes
3. Manage data access, communication, and synchronization

- *Note*: work includes → computation, data access, and I/O

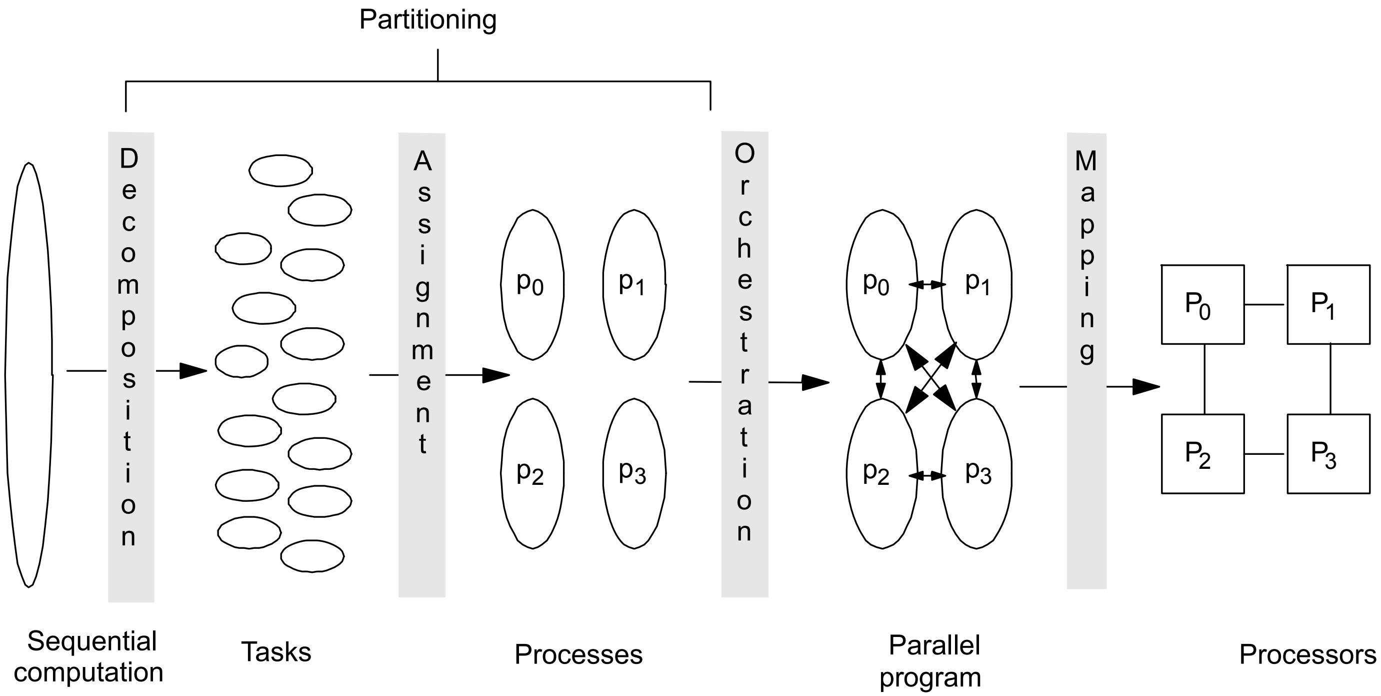
Main goal: Speedup (plus low programming effort and resource needs)

$$Speedup\ (p) = \frac{Performance(p)}{Performance(1)}$$

Some Important Concepts

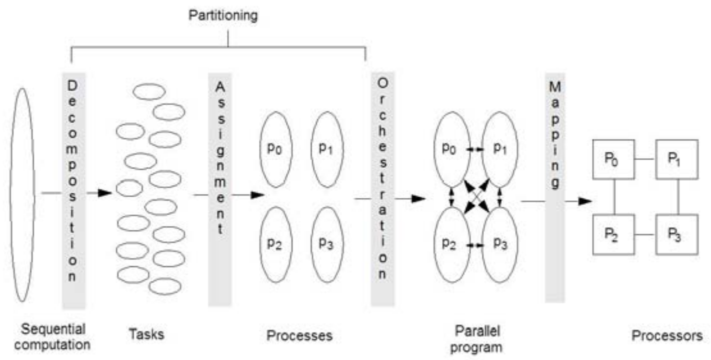
- ***Task***:
  - Arbitrary piece of undecomposed work in parallel computation
  - Executed sequentially; concurrency is only *across* tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
- ***Process (thread)***:
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks
- ***Processor***:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - first write program in terms of processes, then map to processors

Steps in Creating a Parallel Program



- **4 steps: Decomposition, Assignment, Orchestration, Mapping**
  - Done by programmer or system software (compiler, runtime, ...)
  - Issues are the same, so assume programmer does it all explicitly
  - An iterative process!

Decomposition



- ***Decomposition*** ⇔ Process of breaking up a computation into tasks (later to be divided among processes)

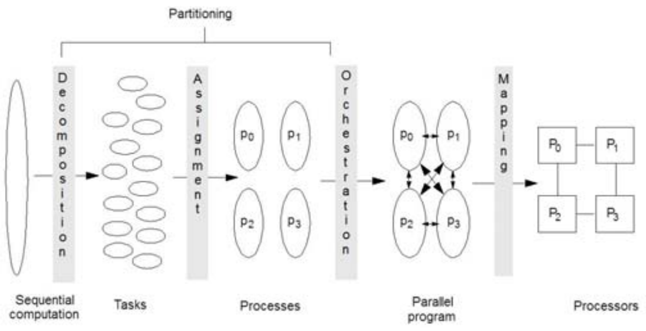
- Since …
  - *Tasks may become available statically or dynamically*
    - *Ex. Static: standard partition of a dense matrix multiply*
    - *Ex. Dynamic: event-based simulation*
  - *Number of available tasks may vary with time (or not)*

… need to identify concurrency (independence) and decide level at which to exploit it

- Goal: Enough tasks to keep processors busy, but not too many

- Recall: Number of tasks available at-a-time is upper bound on achievable speedup

Assignment



Assignment ⇔ Specification mechanism to divide up work (tasks) among processes

– E.g. which process computes forces on which stars, or which rays
– Together with **decomposition**, also called ***partitioning***

Performance goals:

1. Balance workload
2. Reduce communication and management costs

Note: inherent conflict between these goals!

Structured approaches usually work well

– Code inspection (parallel loops) or understanding of application
– Well-known heuristics
– *Static* versus *dynamic* assignment

As programmers, we worry about partitioning first

– *Usually* independent of architecture or programming model
– But cost and complexity of using primitives may affect decisions
– Often requires iterative refinement

Partitioning for Performance

Partitioning ⇔ ***Decomposition + Assignment***

Three major points:

1. Balancing the workload and reducing wait time at synch points
   - *much more later – together account for CPU idle time*
2. Reducing inherent communication
3. Reducing “extra” work
   - *e.g., for load balancing!*

Even these algorithmic issues trade off:

- Minimize comm. ➔ run all on 1 processor ➔ extreme load imbalance
- Maximize load balance ➔ random (low cost) 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

Reducing Inherent Communication

For now, communication is between processes. We won't know about inter-node communication until after mapping processes to processors.

Communication is expensive! ***>> why?***

First → Measure: ***communication to computation ratio***

- *Analogous to computational intensity*
- *Determined by algorithm and assignment of tasks to processes*

Easy part → Assign tasks that access same data to same process

- *Nowadays not always best – **vertical** locality may be as important as **horizontal***

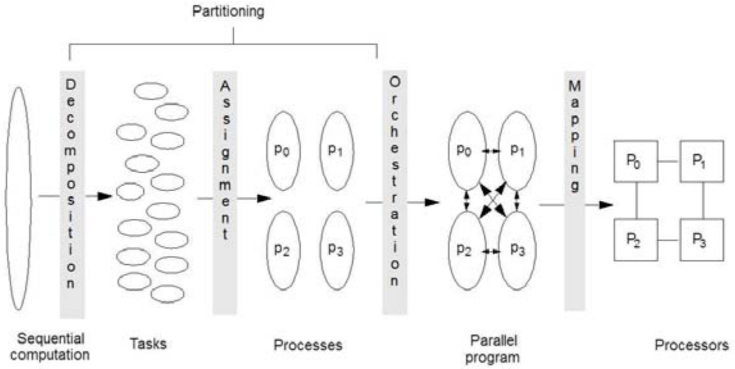
Problem → Data are used not in isolation but with other data

Hard part → Given complete knowledge of inter-task communication, solving communication and load balance is NP-hard in general case

But in real applications → simple heuristic solutions often work well

- *Applications have structure!*
- *Or at least their own problem-specific solutions*

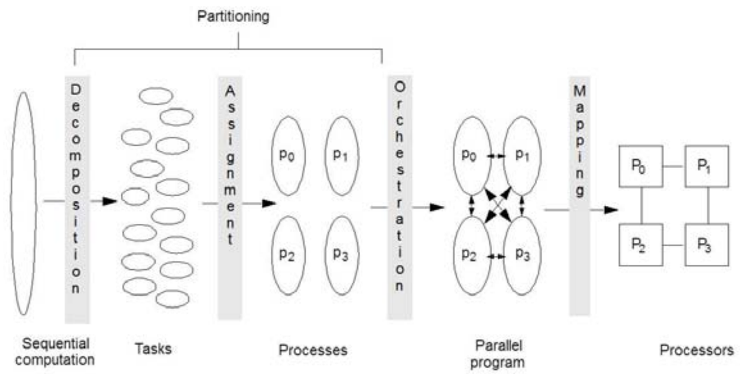
Orchestration



Orchestration ⇔ How processes interact: communication, synchronization, scheduling, etc.

- Orchestration uses available mechanisms (e.g., hardware support, system calls, etc.) for:
  - Naming data (i.e. data access)
    - How local and remote data are accessed → e.g., communication ops
  - Structuring communication
  - Synchronization
  - Organizing data structures and scheduling tasks temporally (to exploit locality)
- Goals (good performance!)
  - Reduce cost of communication and synch. as seen by processors
  - Preserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Tradeoffs
  - Waiting time versus load imbalance
  - More/less communication versus fewer/more processes

Mapping



Mapping ⇔ How processes are assigned to processors

- After orchestration, have parallel program

- Two aspects of mapping:
  1. Which processes will run on same processor, if necessary
  2. Which process runs on which particular processor
     - mapping to a network topology

System Issues –

- Single application or multiple? Single or multiple users? Are processes pinned to processors?
- Control by OS? User space thread manager? "Run-time"?

- One extreme: *space-sharing* a cluster
  - Machine divided into subsets, only one app at a time in a subset
  - Processes can be pinned to processors, or left to OS
- Another extreme: complete resource management control to OS
- Real world is often between the two
  - User specifies desires in some aspects, system may ignore

Default → Usually adopt the obvious view: process ⇔ processor (thread ⇔ core)

Part 4: Back to SOR: Simplify Even Further

Illustrate parallel code →

- *no red-black, simply ignore dependences within sweep*
- *sequential order same as original, parallel program nondeterministic*

“Asynchronous” version of code

```
while (!done) do                       /*a sequential loop*/
  diff = 0;
  for_all i ← 1 to n do                /*a parallel loop nest*/
    for_all j ← 1 to n do
      temp = A[i,j];
      A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
        A[i,j+1] + A[i+1,j]);
      diff += abs(A[i,j] - temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
end while
```

- **`for_all`** removes ***all*** dependencies, leaves assignment to system

Decomposition 1: into elements: *each inner-loop iteration is independent!*

- *degree of concurrency* = n^2 →*TOO MUCH* concurrency. Every operation must be locked to prevent incorrect execution.

Decomposition 2: by rows (get rid of one **`for_all`**)

- *Multiple variations, see next page*

Partitioning – Practical Parallelism w/ strips

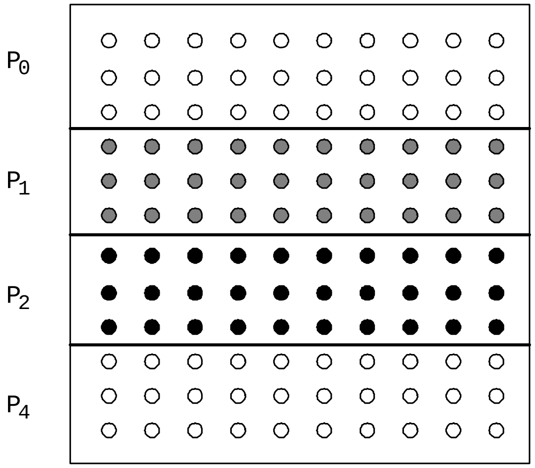
To decompose into rows, make line 18 loop sequential; *n-way* parallel

- but implicit global synch. at end of `for_all` loop

Static assignment by rows reduces concurrency (from n^2 to p)

Static assignments (given decomposition into rows)

- **block assignment** of rows: Row ***i*** is assigned to process
  - ***block*** assignment reduces communication by keeping adjacent rows together
- **cyclic assignment** of rows: Process ***i*** is assigned rows ***i, i+p***, and so on



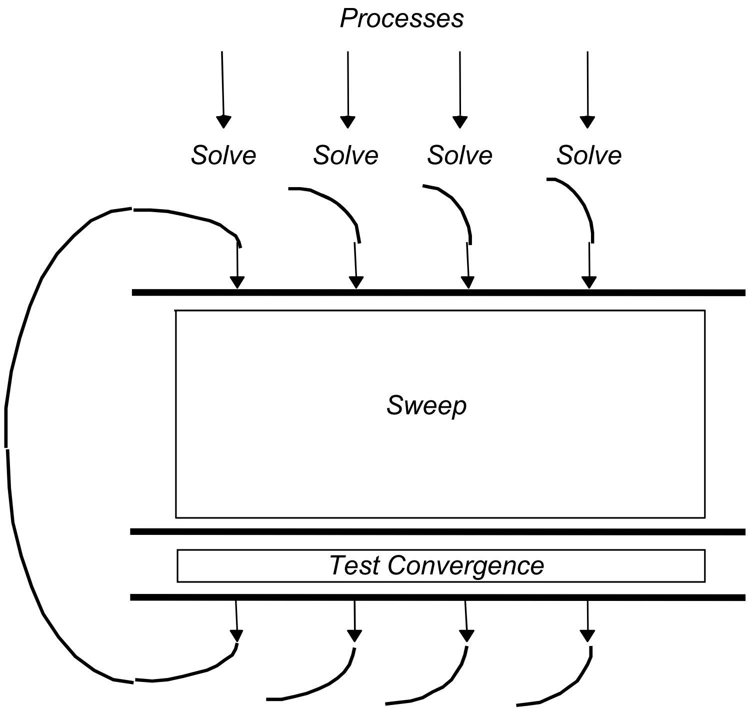
$$\left\lfloor \frac{i}{p} \right\rfloor$$

Also possible – ***Dynamic assignment:*** for each thread →

- get a row index, work on the row, get a new row, and so on

Shared Address Space Solver

Single Program Multiple Data (SPMD)



Note:

→ 2 phases, Sweep and Test

→ Indicates three barriers

** all really needed?*

Assignment controlled by values of variables used as loop bounds

```
      int n, nprocs;                          /*matrix dimension and number of processors to be used*/
      float **A, diff;                        /*A is global (shared) array representing the grid*/
                                              /*diff is global (shared) maximum difference in current
                                              sweep*/
      LOCKDEC(diff_lock);                     /*declaration of lock to enforce mutual exclusion*/
      BARDEC (bar1);                          /*barrier declaration for global synchronization between
                                              sweeps*/

   main()
   begin
         read(n); read(nprocs);               /*read input matrix size and number of processes*/
         A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
         initialize(A);                       /*initialize A in an unspecified way*/
         CREATE (nprocs-1, Solve, A);
        Solve(A);                             /*main process becomes a worker too*/
         WAIT_FOR_END (nprocs-1);             /*wait for all child processes created to terminate*/
   end main

   procedure Solve(A)
      float **A;                                        /*A is entire n+2-by-n+2 shared array,
                                                        as in the sequential program*/
   begin
      int i,j, pid, done = 0;
      float temp, mydiff = 0;                           /*private variables*/
      int mymin = 1 + (pid * n/nprocs);                 /*assume that n is exactly divisible by*/
      int mymax = mymin + n/nprocs - 1                  /*nprocs for simplicity here*/

      while (!done) do                            /*outer loop until convergence*/
         mydiff = diff = 0;                       /*set global diff to 0 (okay for all to do it)*/
         BARRIER(bar1, nprocs);                   /*ensure all reach here before anyone modifies diff*/
         for i ← mymin to mymax do                /*for each of myrows*/
            for j ← 1 to n do                     /*for all nonborder elements in that row*/
               temp = A[i,j];
               A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                  A[i,j+1] + A[i+1,j]);
               mydiff += abs(A[i,j] - temp);
            endfor
         endfor
         LOCK(diff_lock);                         /*update global diff if necessary*/
         diff += mydiff;
         UNLOCK(diff_lock);
         BARRIER(bar1, nprocs);                   /*ensure all reach here before checking if done*/
         if (diff/(n*n) < TOL) then done = 1;                /*check convergence; all get same answer*/
         BARRIER(bar1, nprocs);
      endwhile
   end procedure
```

Notes on SAS Program

SAS ⇔ Shared Address Space

- SPMD: not lockstep or even necessarily same instructions
- Assignment controlled by values of variables used as loop bounds
  - unique pid per process, used to control assignment
- **Done** condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?

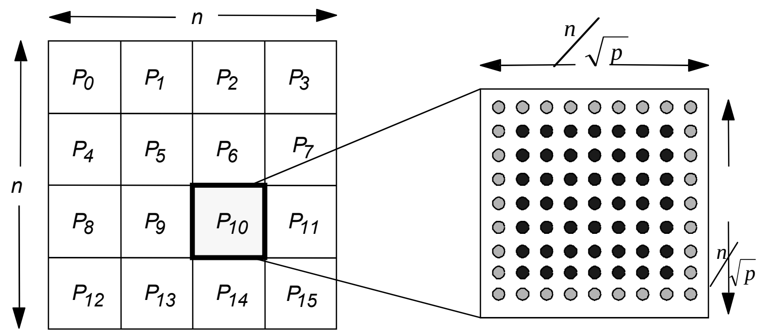
Generalize: Domain Decomposition

- 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

Metric: Try to minimize communication to computation ratio

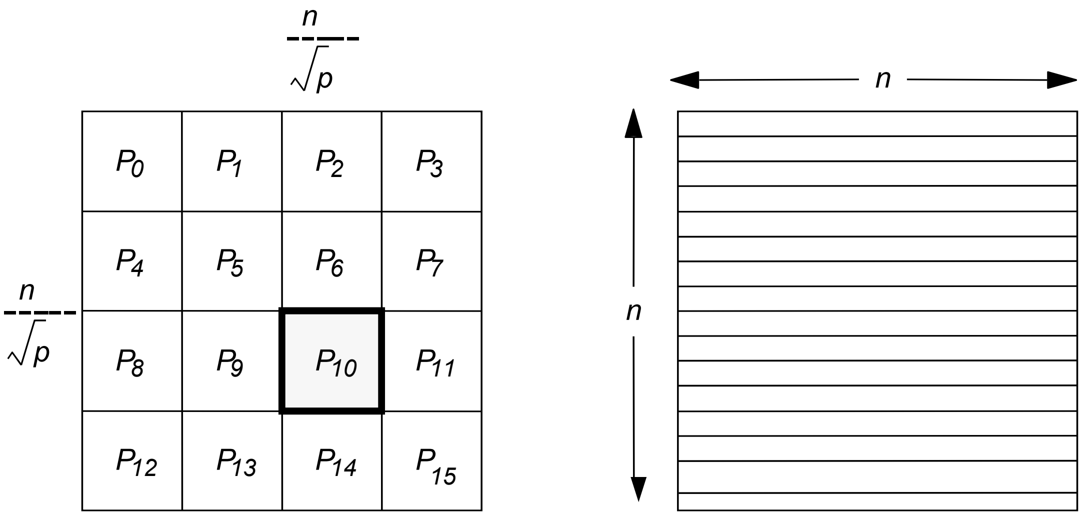


Perimeter to Area comm-to-comp ratio (area to volume in 3-d)

•Depends on n,p: decreases with n, increases with p

Domain Decomposition *(cont.)*

Best domain decomposition depends on comm/comp requirements
Nearest neighbor example: block versus strip decomposition:



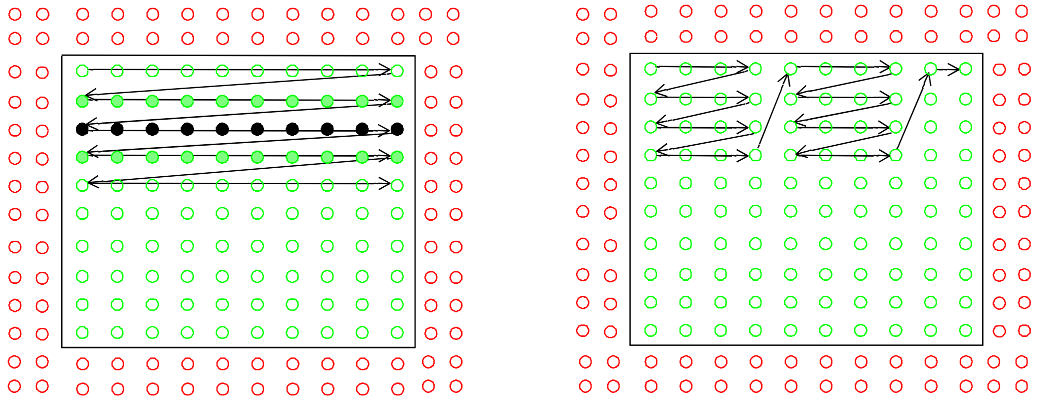
For both decompositions: computation per processor = $\boldsymbol{n^2/p}$

Then → Comm to comp: $\frac{4*\sqrt{p}}{n}$ for block, $\frac{2*p}{n}$ for strip

- Why?
  - *Application dependent: strip may be better in other cases*
    - *E.g. particle flow in tunnel*

Orchestration: Exploiting *Temporal* Locality

- Structure algorithm so working sets map well to memory 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
  - *How much reuse does this get us?*



(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 (higher computational intensity)

– *many linear algebra computations → factorization, matrix multiply*

Orchestration: Exploiting *Spatial* Locality

- Besides capacity, granularities are important:
  - Granularity of allocation
  - Granularity of communication or data transfer
  - Granularity of coherence

Artifactual Communication ≡ Additional communication (not required to solve problem) that is an artifact of the HW/SW implementations

- 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

Spatial Locality Example

Application → Repeated sweeps over 2-d grid, each time adding 1 to elements

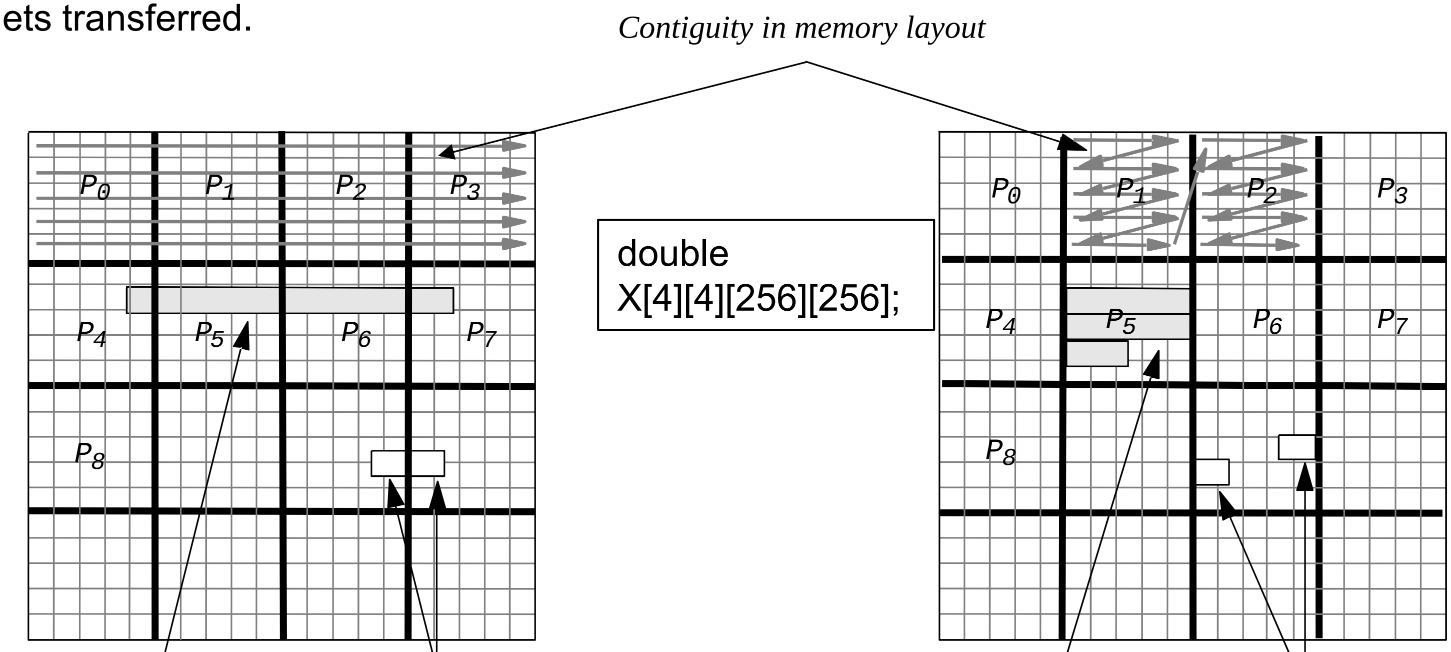
- Natural 2-d versus higher-dimensional array representation

Hardware Model

- Physically distributed memory
- Single address space
- Data transfer granularity = page.
  → That is, if you need another processors data, a whole page gets transferred.
- 4KB page
- P = 16

Data = 1K x 1K array of doubles
1K x 1K x 8B = 8 MB = 2K pages
128 pages per processor

```
double
X[1024][1024];
```



```
double
X[4][4][256][256];
```

Page straddles partition boundaries: difficult to distribute memory well

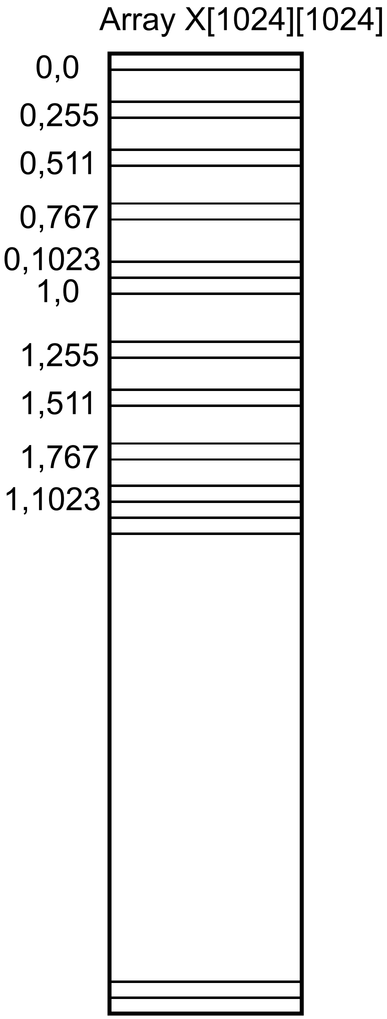
Cache block straddles partition boundary

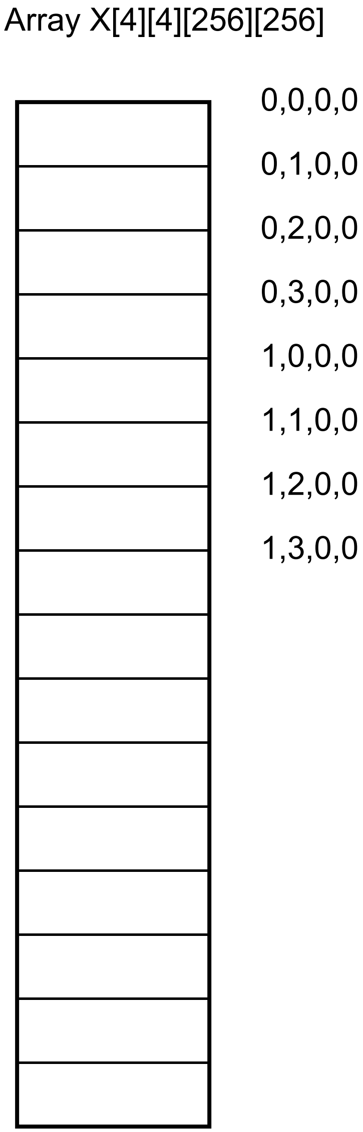
(a) Two-dimensional array

Page does not straddle partition boundary

Cache block is within a partition

(b) Four-dimensional array


Array X[1024][1024]
0,0
0,255
0,511
0,767
0,1023
1,0
1,255
1,511
1,767
1,1023

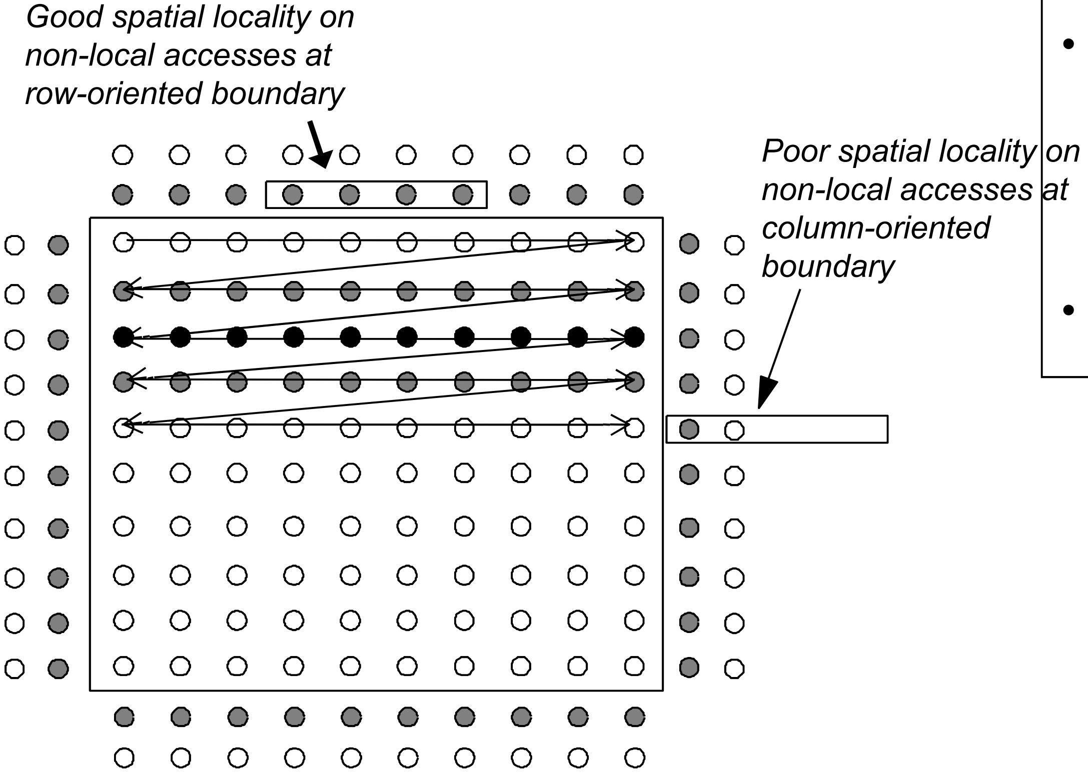

Array X[4][4][256][256]
0,0,0,0
0,1,0,0
0,2,0,0
0,3,0,0
1,0,0,0
1,1,0,0
1,2,0,0
1,3,0,0

Tradeoffs with Inherent Communication

- Partitioning grid solver: blocks versus rows
  - Blocks still have a spatial locality problem on remote data
  - Row-wise can perform better despite worse inherent c-to-c ratio

Model: Assume Block fits in cache

- neighboring Blocks in cache of other processors
- Write by another processor invalidates your copy (it's now stale)
- No need to worry about cache coherence mechanism: suffice that on every iteration, the first read to a cache block outside your own Block will always be a miss.
- What's better, Blocks or Strips?



Good spatial locality on non-local accesses at row-oriented boundary
Poor spatial locality on non-local accesses at column-oriented boundary

- *Result depends on p*

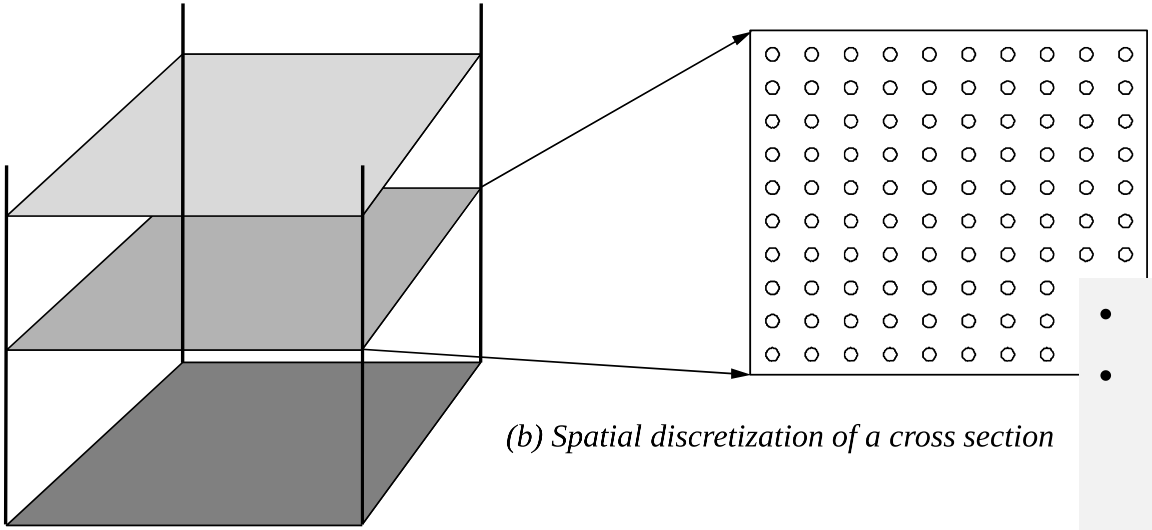
Motivating Problems

1. Simulating Ocean Currents
   - Regular structure, scientific computing
2. Simulating the Evolution of Galaxies
   - Irregular structure, scientific computing
3. Rendering Scenes by Ray Tracing
   - Irregular structure, computer graphics

- What they all have in common: substantial available parallelism

- Differences: various aspects of regularity

Case Study 1: Simulating Ocean Currents



(a) Cross sections

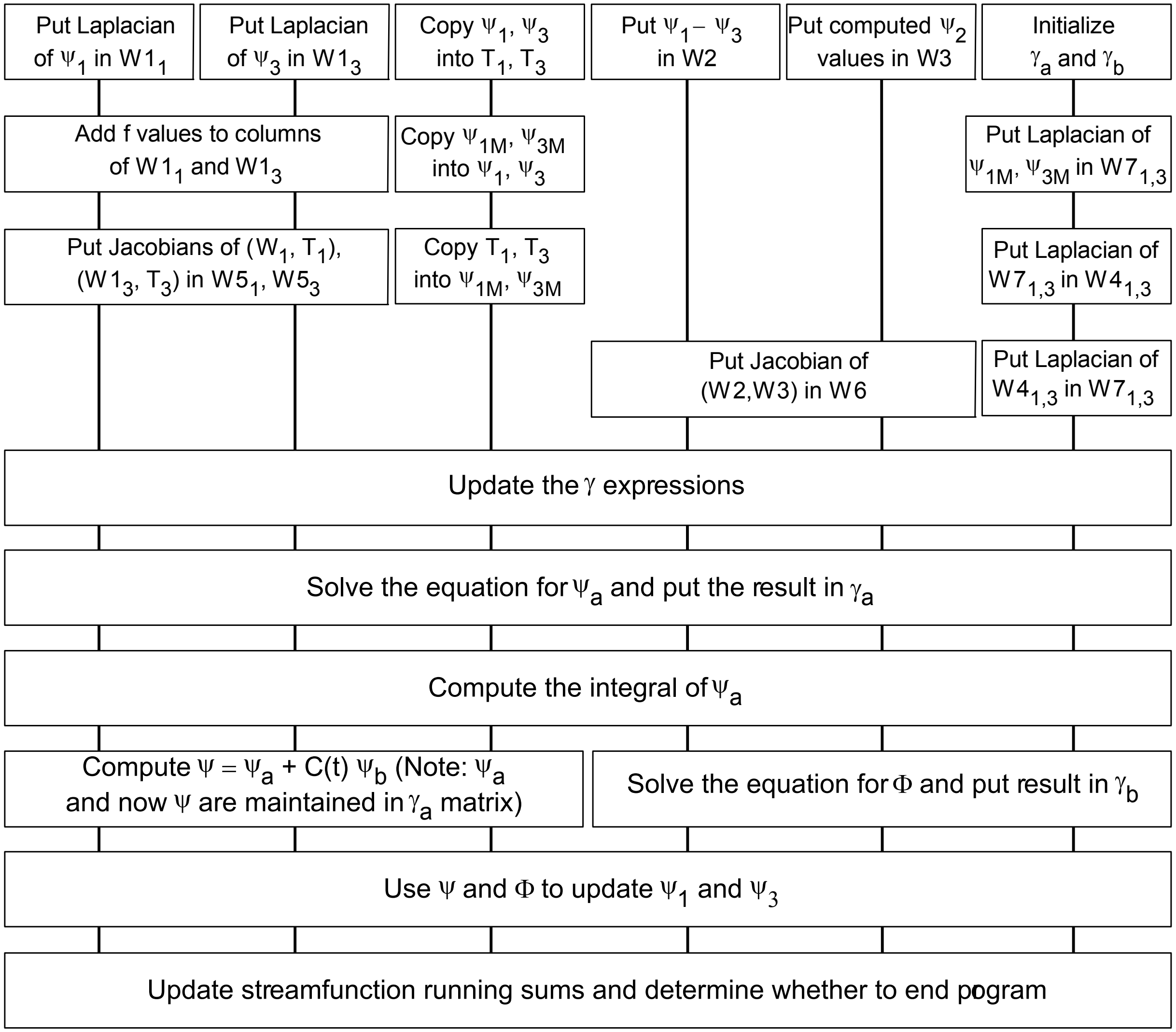
(b) Spatial discretization of a cross section

- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations
- (mostly) Local data exchange

- Simulates currents in an ocean basin
- At each horizontal cross section, several variables are modeled: current, temperature, pressure, friction, etc. (25 grid structures)
- Each time step consists of 33 grid computations: combining and performing sweeps.
- Sweeps are done using *Multigrid Method:*
  - Hierarchy of grids: n x n, n/2 x n/2, n/4 x n/4, etc.
  - Successive sweeps are performed on coarser grids
  - Can return to finer grids depending on diffs computed on previous sweep.
  - Done when diffs on finest grid is < epsilon.

Ocean Intra-Time-Step Structure

- 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 pogram

Partitioning

- Exploit data parallelism only
  - 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

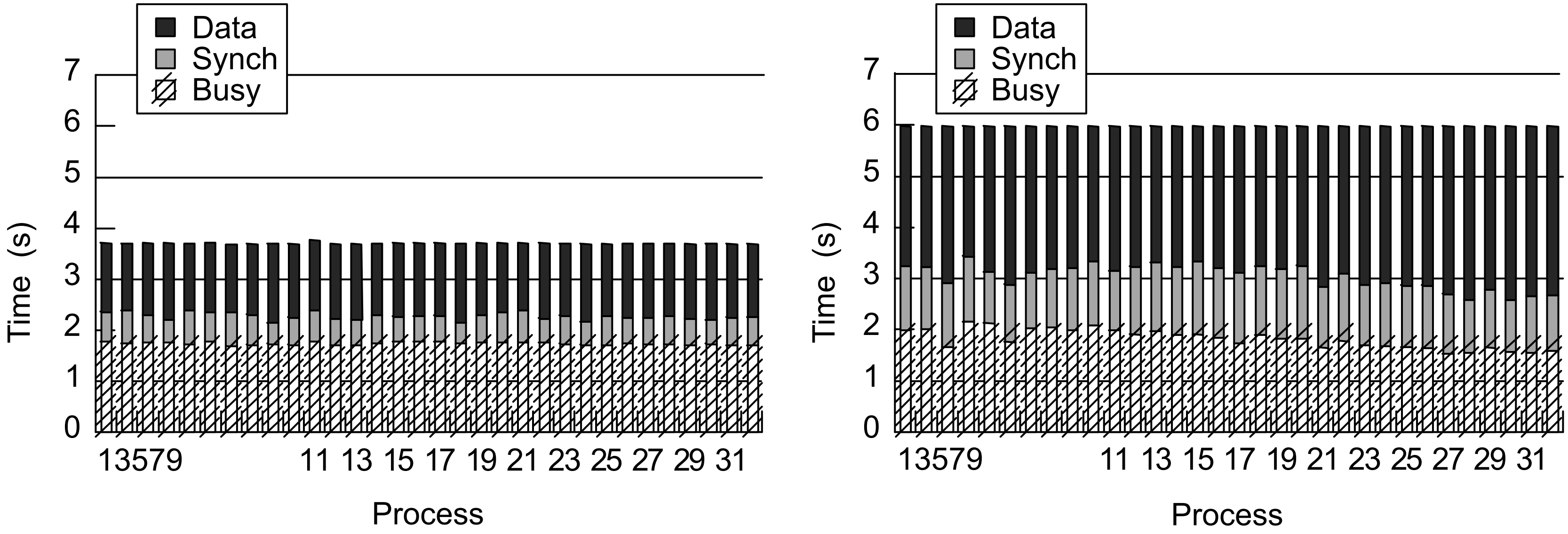
- Complex working set hierarchy
  - (i) A few points for near-neighbor reuse, (ii) three sub-rows, (iii) partition of one grid, (iv) partitions of multiple grids in a hierarchy, (v) some data across iterations, (vi) all data
  - First three or four most important
  - Large working sets, but data distribution 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

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



(a) 4-d grids ***(b) 2-d grids***

– 4-d grids much better than 2-d, despite very large caches on machine

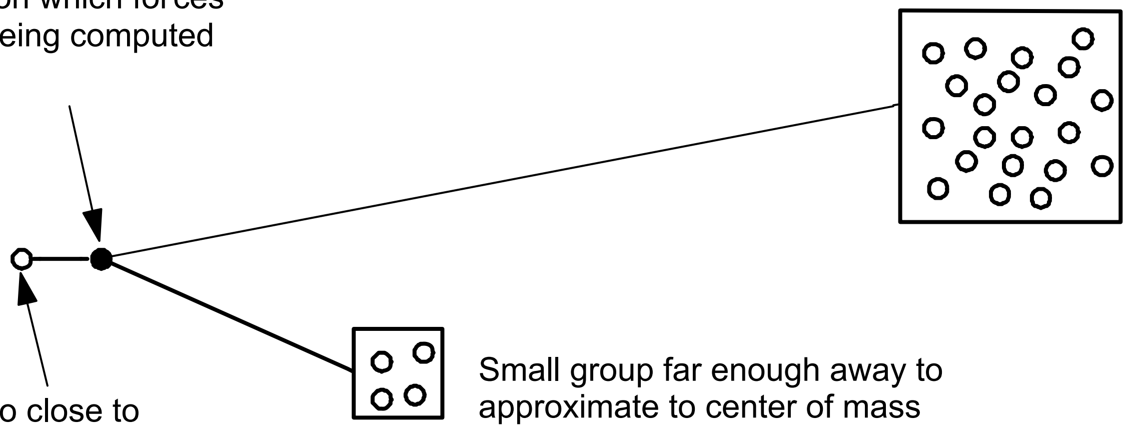
- data distribution is much more crucial on machines with smaller caches

– Major bottleneck in this configuration is time waiting at barriers

- imbalance in memory stall times as well

Simulating Galaxy Evolution

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



Star on which forces are being computed

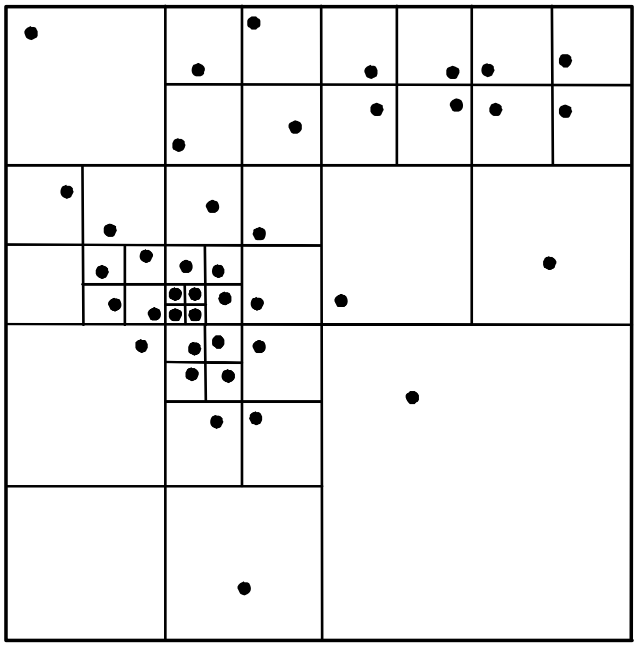
Large group far enough away to approximate

Star too close to approximate

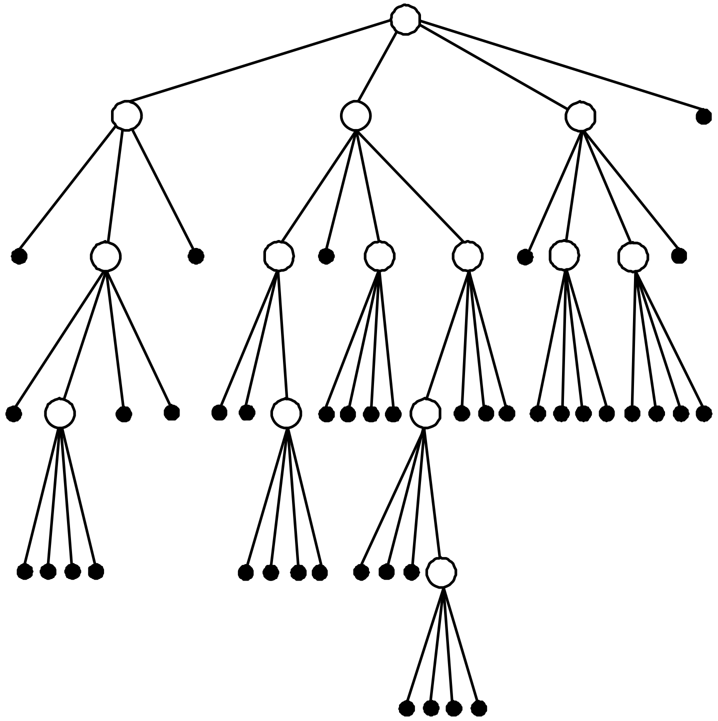
Small group far enough away to approximate to center of mass

- *Many time-steps, plenty of concurrency across stars*
- *Non-uniform data structures*

Case Study 2: Barnes-Hut



(a) The spatial domain

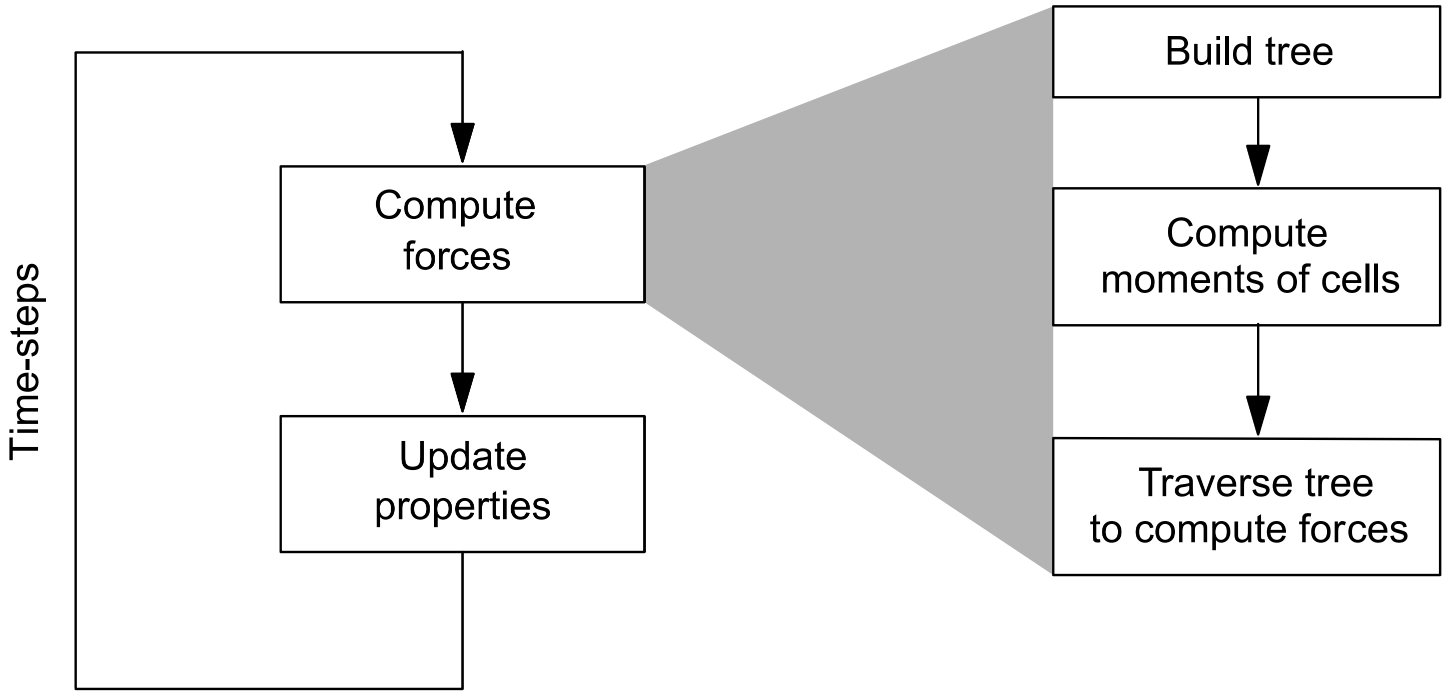


(b) Quadtree representation

- 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

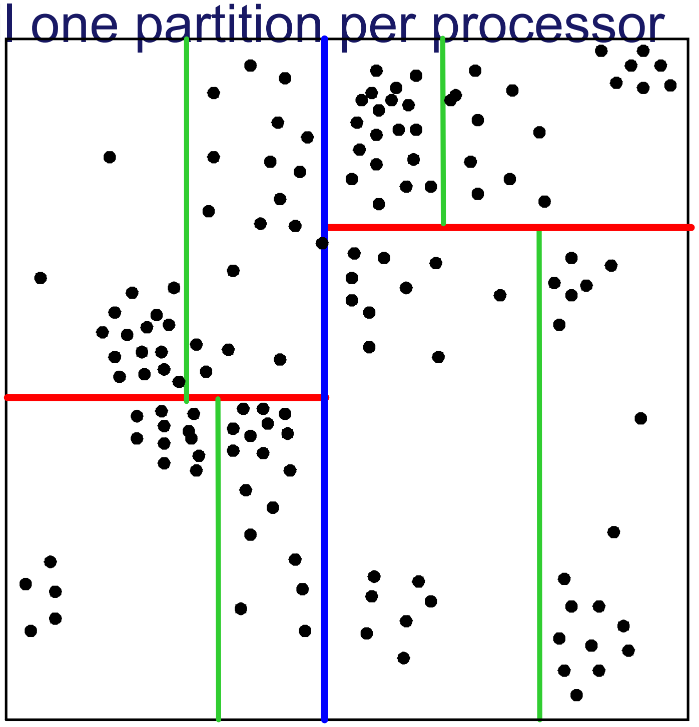
- Equal particles ≠ equal work.
  - Solution: Assign costs to particles based on the work they do

- 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

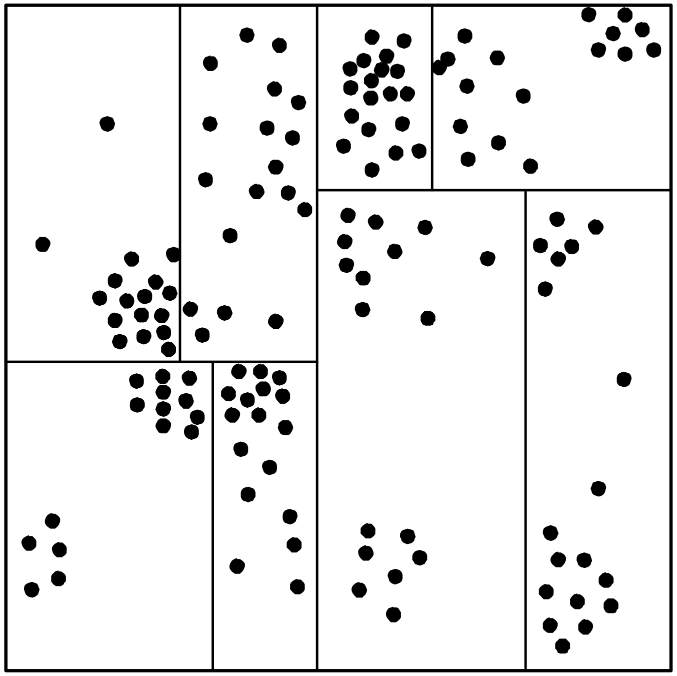
- Orthogonal Recursive Bisection:
  - Recursively bisect space into subspaces with equal work
    - Work is associated with bodies, as before
  - Continue until one partition per processor



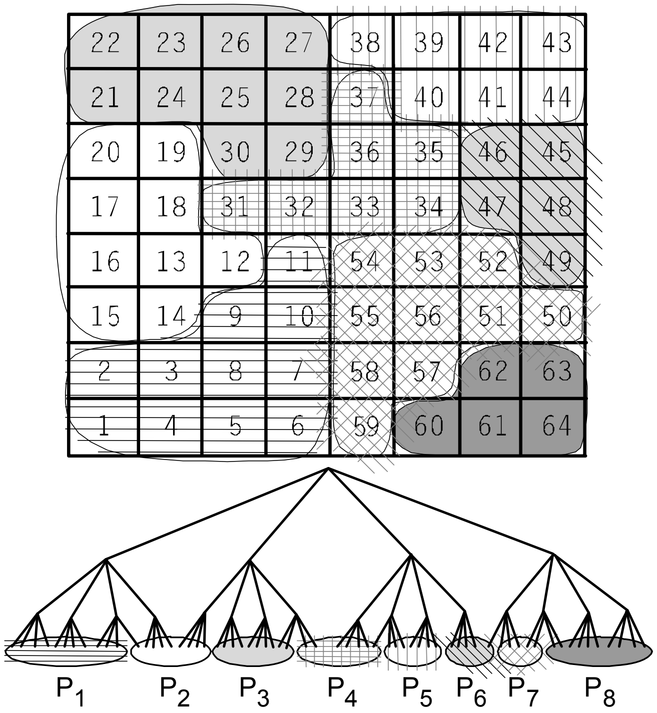
- *High overhead for large no. of processors*

Another Approach: Costzones

- Insight: Tree already contains an encoding of spatial locality.



(a) ORB



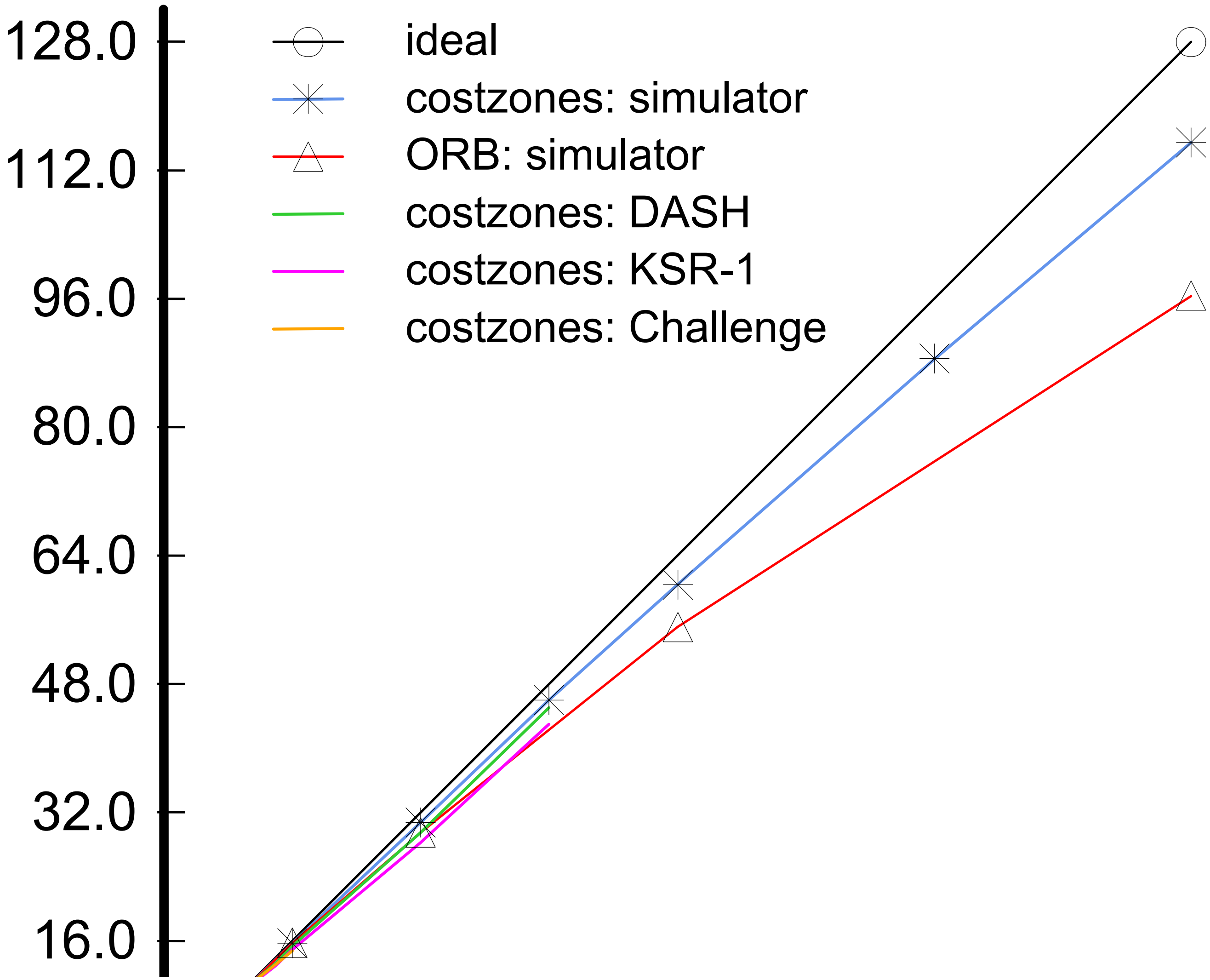
22	23	26	27	38	39	42	43
21	24	25	28	37	40	41	44
20	19	30	29	36	35	46	45
17	18	31	32	33	34	47	48
16	13	12	11	54	53	52	49
15	14	9	10	55	56	51	50
2	3	8	7	58	57	62	63
1	4	5	6	59	60	61	64

(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

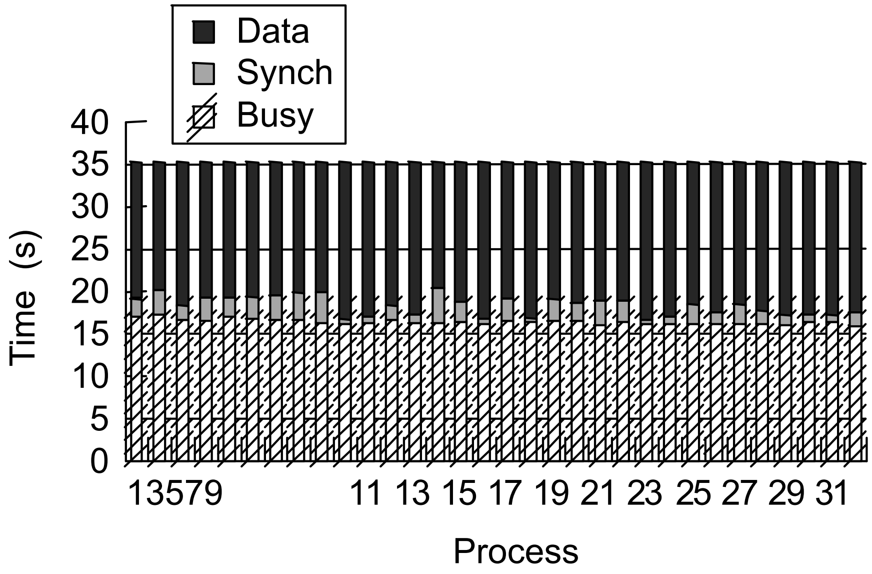


Orchestration and Mapping

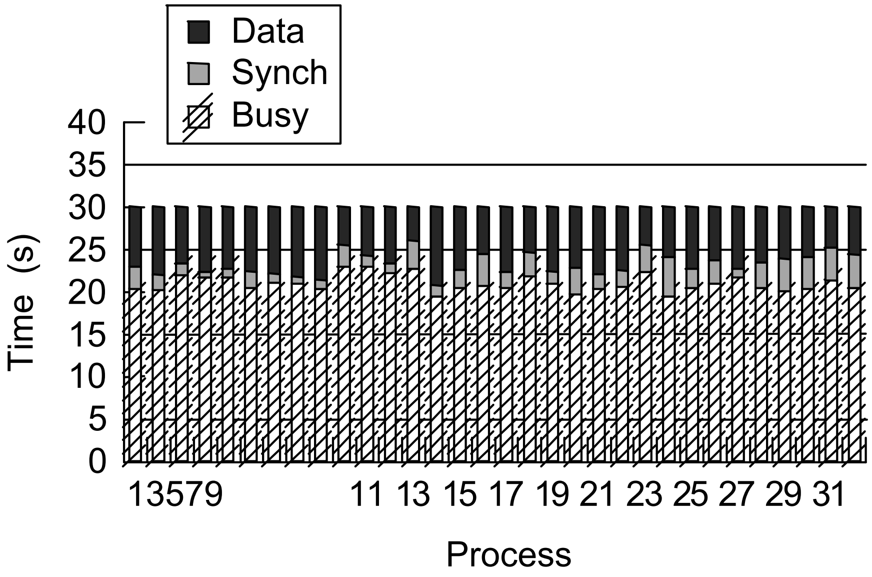
- Spatial locality: Very different than in Ocean, like other aspects
  - Data distribution is much more difficult than
    - Redistribution across time-steps
    - Logical granularity (body/cell) much smaller than page
    - Partitions contiguous in physical space does not imply contiguous in array
    - 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 cost zones*



(a) Static assignment of bodies



(b) Semistatic costzone assignment

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

Rendering Scenes by Ray Tracing

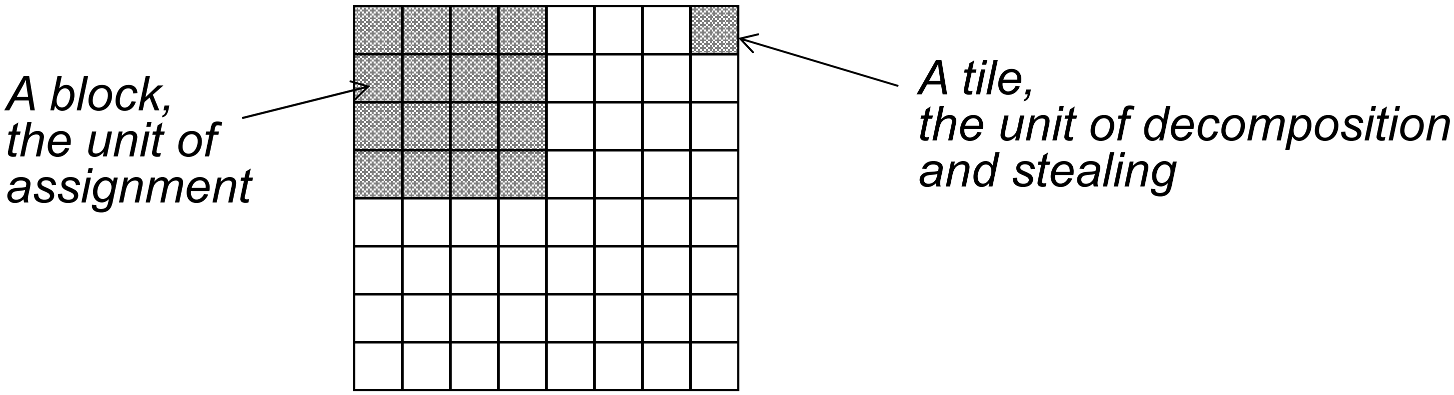
- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays
- Work per ray can vary tremendously

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

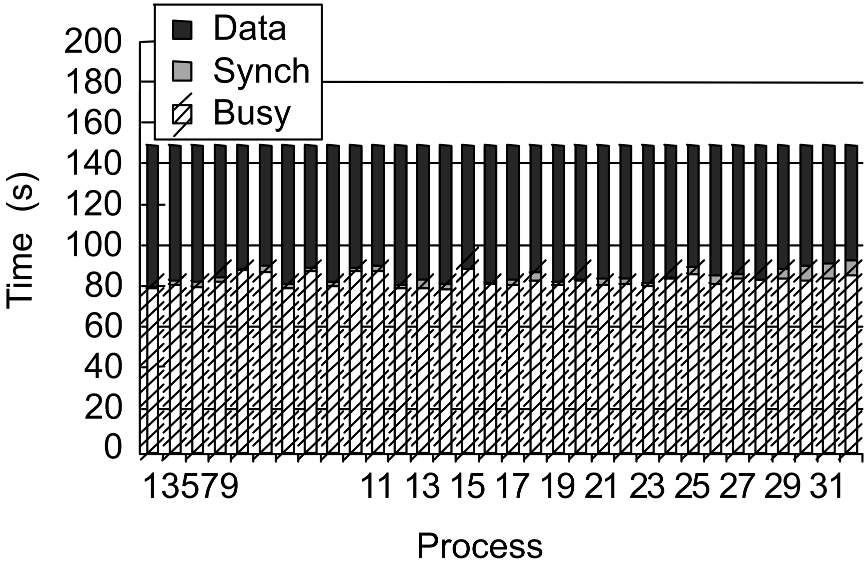


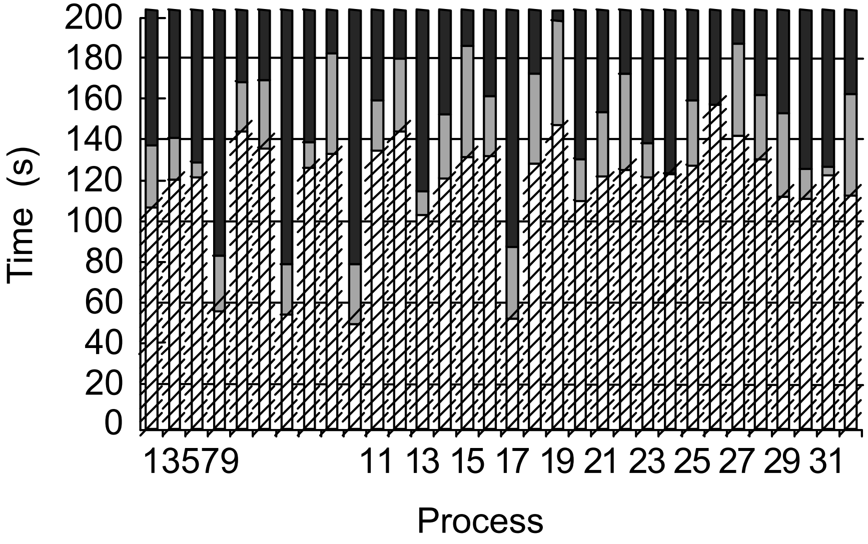
Could use 2-D interleaved (scatter) assignment of tiles instead

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