

11 prog

# Intro to Parallel Programming

- Function units
- Pipeline function units
- vector extensions

ILP & SIMD

MIMD - multi-threaded  
Programming

Diffs Application Space - Data Parallel Programs

1) Single/multi thread - way of thinking

2) Historic - convergence of technologies { CPU, Vector }  
MPP - cluster of CPUs

3) Tech - Multicore - Different techniques - Threads

4) Scale - Large Proc. Large Problems

# Intro to Parallel Programming

- bit parallel

- function /  
LD  
ADD  
Floats.

- pipeline -

- Vector

Thread level parallelism



multiple cores

# Intro to Parallel Programming

## Outline

1. Intro – Why parallel computers?
2. Why is parallel programming  $\neq$  serial programming?
  - Extracting ILP
  - Vectorization – what does vectorizable code need to look like?
  - Parallelizing Compilers
  - 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).



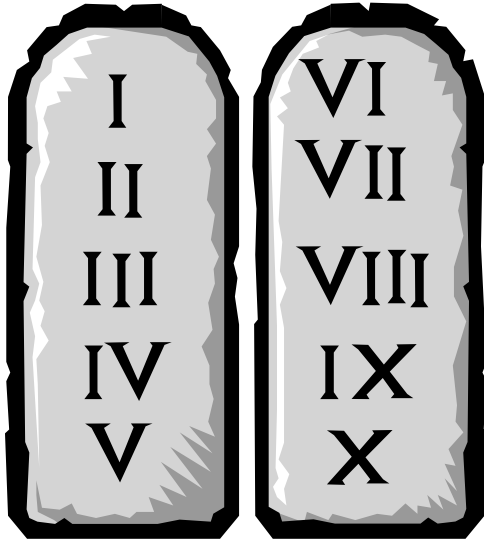
*Holy grail of parallel computer architecture:*

*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?



# 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?
  - $P = 2 \Leftrightarrow 18$  months,  $P = 4 \Leftrightarrow 3$  years, etc.

→ *Aside: Is Moore's law failing? Yes ...*

- 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: 50+ years of work in parallelizing compilers.

Sketch of approach: Analyze and optimize program.

- Find dependencies (“hazards” in ILP) – whatever instructions are not dependent can be executed in parallel
- Schedule code in basic blocks (pipeline, fill in bubbles)
- Loop transformations – interchange, blocking
- Loop unrolling (also - splitting, fusion)
- Strip mining

***Use methods outlined in next section, but automate***

# Try Sequential Code?

## Fundamental Challenge:

- 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 (or the programming language) will often introduce unnecessary dependencies that force them to be executed sequentially.***

**Dependencies:** True (data) dependencies occur when one instruction needs data computed by another instruction (RAW). For pipelined and superscalar data paths, we are able to schedule instructions after identifying the data dependencies – see example below. This improves performance by removing stalls. Loop unrolling also helps because it gives more latitude to the code scheduler.

What's different now - with parallel architectures?

# Data Dependence

**Data Dependence**  $\Leftrightarrow$  An ordering constraint. An ordering of operand usage such that changing that ordering will cause incorrect program execution.

Example: Clearly i0 must be executed before i1,i2,i3,i4 But i1,i2,i3,i4 can be executed in any order. Or in parallel.

```
# $2 initial value = 20
i0:  sub    $2,$1,$3      # reg 2 written w/ -10
i1:  and    $12,$2,$5     # reg 2 read
i2:  or     $13,$6,$2     # reg 2 read
i3:  add    $14,$2,$2     # reg 2 read
i4:  sw     $15,100($2)   # reg 2 read
```

# Data (RAW) dependence example from datapath

**RAW (Read After Write) Dependence** ⇔ A later instruction reads an operand that was written by a previous instruction.

**RAW Hazard** ⇔ The instructions are exchanged or otherwise interleaved, as through pipelining, such that the read occurs before the write causing incorrect execution.

*Note that the pipeline presents a special case. Even instructions executed in program order – exactly as written – could execute incorrectly if not handled properly. E.g., with forwarding or reordering.*

The dependence (original code) *indicated by arrow*:

I1: OR **R1**,R7,R8

I2 - I6: ... some instructions that do not use R1

I7: ADD **R1**,R2,R3 # I8 needs **R1** from I7 not I1

I8: SUB R5,  **R1**,R4

The hazard:



*Note: Other kinds are WAW and WAR – they are B.T.S.O.T.C*

# Dependences versus hazards versus stalls/bubbles

Data Dependence ⇔ An ordering constraint. That is, an order of operand usage such that changing that ordering will cause incorrect program execution.

Hazard ⇔ a situation (code/HW pair) that prevents an instruction stage from proceeding

Stall/Bubble ⇔ In a pipeline, when an instruction cannot proceed, it is held up for a cycle (or more). It is said to stall. In it's place a NOP/bubble is inserted.

- There is nothing necessarily wrong with dependences. They are a part of most programs and are often unavoidable. *As we shall see, however, writing code with fewer dependences can sometimes improve performance.*
- A dependence becomes a performance problem if it prevents us from parallel execution, either in a pipeline, or a vector, or in multiple function units, or in multiple cores, or in multiple CPUs.

# 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 a computer architecture (for good reasons). *Naïve* translation is likely to introduce inefficiencies into machine code.

**Goal of Optimization** → Eliminate these inefficiencies.

## **Optimization Procedure** →

- Program is a specification not a recipe. Can be transformed as long as it executes correctly >> **ForAll 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  $\langle S_i, S_j, \dots, S_z \rangle$  denote a dependence relation among instructions. That is,  $S_i$  must execute before  $S_j$  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  $\langle S_1, S_2, S_3 \rangle$

But clearly  $\langle S_2, S_1, S_3 \rangle$  is also correct.

So  $\langle S_1, S_3 \rangle$  and  $\langle S_2, S_3 \rangle$  are dependences but  $\langle S_1, S_2 \rangle$  is not.

Valid transformation:

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

Why do we care? Because *the transformed code may be more efficient*. For example, it could fill in a slot in the pipeline that would otherwise be a stall (bubble).

# A Statically Scheduled CPU

Before continuing, let's look at code scheduling. We start with the original 5-stage CPU. It issues at most one instruction per cycle.

- Single instruction issued per cycle (may be less)
- If instruction after LW uses new data, need to stall
- Pad an unused instruction with NOP
- Compiler can remove some/all hazards
  - Reorder instructions for efficiency

```
Loop: lw    $t0, 0($s1)      # $t0=array element
      NOP                    # $t0 used by next instruct
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
```

- *Instructions per Cycle (IPC) = 5/6 (peak IPC = 1)*
- *RAW following a LW requires 1 cycle stall*



# A Statically Scheduled CPU

```
# Original code
Loop: lw    $t0, 0($s1)      # $t0=array element
      NOP                    # $t0 used by next instruct
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)     # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
```

Improve the code by scheduling instructions

```
# Improved code
Loop: lw    $t0, 0($s1)      # $t0=array element
      addi  $s1, $s1, -4     # move into NOP slot
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)     # store result
      bne   $s1, $zero, Loop # branch $s1!=0
```

- *Instructions per Cycle (IPC) = 5/5 (peak IPC = 1)*
- *“addi” instruction can be moved into NOP slot*

# A Statically Scheduled *Superscalar* CPU

Now: **two** pipelines (current CPUs have many)

- Two-issue packets (*instructions go two at a time*)
  - One ALU/branch instruction
  - One load/store instruction
  - Pad an unused instruction with NOP
- Compiler must remove some/all hazards
  - Reorder instructions for efficiency
  - Constraint: No dependencies are allowed within a packet
  - Possibly some dependencies between packets (handled w/ forwarding)

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

# Hazards in the dual-issue MIPS

- In comparison with the single issue pipeline, more instructions executing in parallel
- We still have the old hazard following a LD instruction
- But → New: EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - add     `$t0, $s0, $s1`  
    load   `$s2, 0($t0)`
    - Must split these into two packets, effectively adding a stall
- Load-use (data) hazard
  - Still one cycle use latency, but now two instructions
- Therefore: More aggressive scheduling required

# Superscalar Scheduling Example

```

Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
    
```

- *RAW following a LW still requires 1 cycle stall*

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	nop	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	addi \$s1, \$s1, -4	sw \$t0, 4(\$s1)	4
	bne \$s1, \$zero, Loop	nop	5

- $IPC = 5/5 = 1$  (peak  $IPC = 2$ )

# Superscalar Scheduling Example

Loop: lw  $\$t0$ , 0( $\$s1$ ) #  $\$t0$ =array element  
 addu  $\$t0$ ,  $\$t0$ ,  $\$s2$  # add scalar in  $\$s2$   
 sw  $\$t0$ , 0( $\$s1$ ) # store result  
 addi  $\$s1$ ,  $\$s1$ , -4 # decrement pointer  
 bne  $\$s1$ ,  $\$zero$ , Loop # branch  $\$s1 \neq 0$

*independent*

- *This time schedule code*

	ALU/branch	Load/store	cycle
Loop:	nop	lw $\$t0$ , 0( $\$s1$ )	1
	addi $\$s1$ , $\$s1$ , -4	nop	2
	addu $\$t0$ , $\$t0$ , $\$s2$	nop	3
	bne $\$s1$ , $\$zero$ , Loop	sw $\$t0$ , 4( $\$s1$ )	4

- $IPC = 5/4 = 1.25$  (peak  $IPC = 2$ )

# Loop Unrolling (again)

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called *register renaming*
  - Avoid loop-carried *anti-dependencies*
    - Store followed by a load of the same register
    - Aka *name dependence*
      - Reuse of a register name

# Loop Unrolling Example

*Original*

```

Loop: lw    $t0, 0($s1)      # $t0=array element
      addu  $t0, $t0, $s2    # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi  $s1, $s1, -4     # decrement pointer
      bne   $s1, $zero, Loop # branch $s1!=0
  
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

Comparison:

- \* Unrolled, each additional iteration has a marginal cost of 2 cycles.
- \* Not unrolled, dual issue, scheduled: each additional iteration costs 4 cycles.
- \* Original single issue unoptimized costs 6 cycles per iteration.

$$\text{IPC} = 14/8 = 1.75 \quad \text{Peak IPC} = 2$$

# Vector Code & Vectorization

```
// Input code → C = A + B;    // A,B,C are vectors
VLOAD  V1,A                    // Vector assembly language (VAL)
VLOAD  V2,B
VADD    V3,V1,V2
VSTORE C,V3
```

Note: The semantics of this code is **SIMD**.

**SIMD** ⇔ Each instruction operates on a vector of data (say, 64 elements), and each instruction completes before the next begins. At least, **result** must appear as if they do, even if there are HW optimizations such as *chaining*.

*But most Fortran is written like this, even for vector code:*

```
// Example 1: Fortran 77
DO I = 1, 64
    C(I) = A(I) + B(I)
ENDDO
```

*Can serial code be  
rewritten (compiled) into  
vector A.L.*



# Vector Code & Vectorization

Let's start with the vector code in Example 1

```
// Example 1: Fortran 77
DO I = 1, 64
    C(I) = A(I) + B(I)
ENDDO
```

Can we Vectorize it? That is can we convert it to SIMD syntax and still have the same semantics?

Method (sketch): Replace array index with range of array indices as specified in the Loop. This is like MatLab (or Fortran 90).

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

Is Example 1 Vectorizable? Do both versions give the same answer?

Yes! Going from *scalar* syntax to *array* syntax does not change the outcome, so this too can be translated into the same Vector A.L.

**Vectorizable** ⇔ A Fortran 77 code that can be rewritten in Fortran 90 array syntax without changing the semantics (meaning) – still behaves same as the serial version.

// Example 2: What does this code do?

```
DO I = 1, 64
```

```
  A(I+1) = A(I) + B(I)
```

```
ENDDO
```

Is this Fortran 77 code vectorizable? To tell, translate into Fortran 90 array syntax and check output. Hint: each iteration uses a result from the previous iteration.

// Replace index with range given by loop

```
A(2:65) = A(1:64) + B(1:64)
```

For the Serial and Parallel versions – do they give the same answer?

Note: In the parallel version, all inputs are read before any writes and so refer to the original array values. This is different from the code above and so is **not**

**vectorizable.**

A = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
B = 1 2 3 4 5

Shift & sum up by one? ...  
????

= Really - Combining

??

**Vectorizable** ⇔ A Fortran 77 code that can be rewritten in Fortran 90 array syntax without changing the semantics (meaning) – still behaves same as the serial version.

```
// Example 3:  
DO I = 2, 65  
    A(I-1) = A(I) + B(I)  
ENDDO
```

```
// Array version  
A(1:64) = A(2:65) + B(2:65)
```

This code ***is vectorizable!***

# Loop level parallelization (versus SIMD)

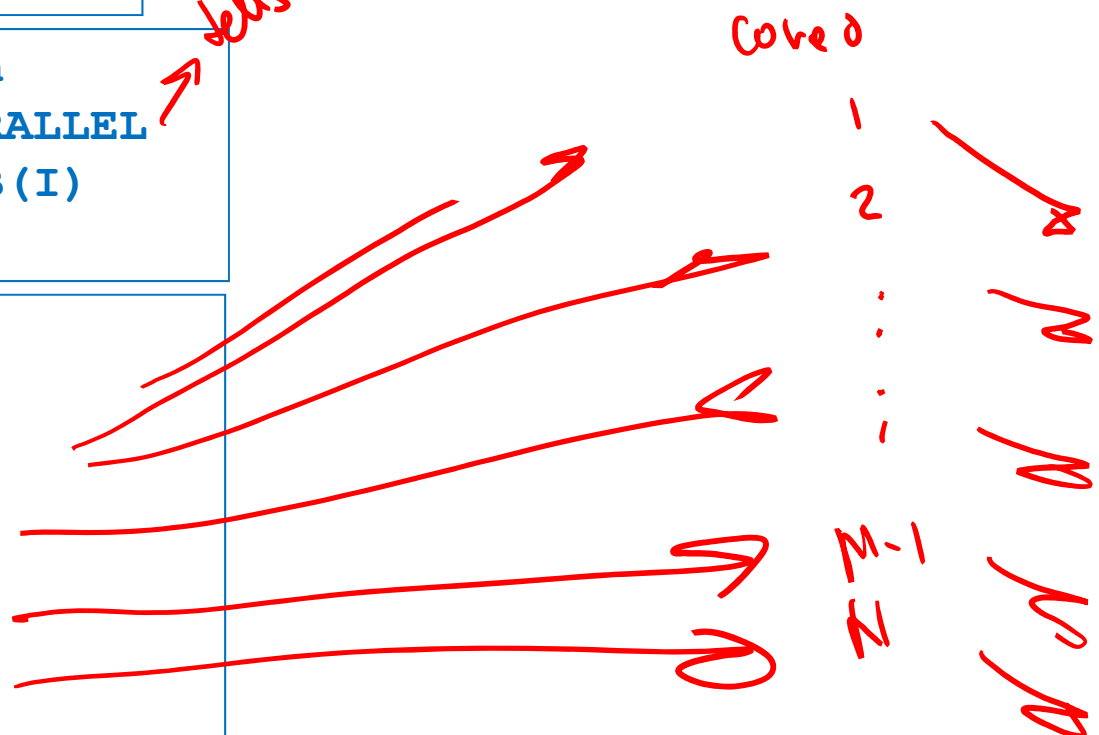
**Parallel execution** → **iterations** can be done in parallel. This assumes that execution is tolerant of iteration reordering.

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

```
// Parallelization  
DO I = 1, N IN PARALLEL  
    A(I) = A(I) + B(I)  
ENDDOINPARALLEL
```

```
// Parallelization  
DO IN PARALLEL  
A(1) = A(1) + B(1)  
A(2) = A(2) + B(2)  
.  
.  
A(N) = A(N) + B(N)  
ENDDOINPARALLEL
```

*tells compiler = no dependence  
potentials*



# Parallelize through data decomposition

**Parallel execution** → iterations can be done in parallel. This assumes that execution is tolerant of iteration reordering.

**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? Yes  
DO I = 1, N  
    A(I) = A(I) + B(I)  
ENDDO
```

```
// Parallelizable? No #3  
DO I = 1, N  
    S = A(I) + B(I)  
ENDDO
```

A&K

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

```
// Parallelizable? No  
DO I = 1, N  
    A(I-1) = A(I) + B(I)  
ENDDO
```

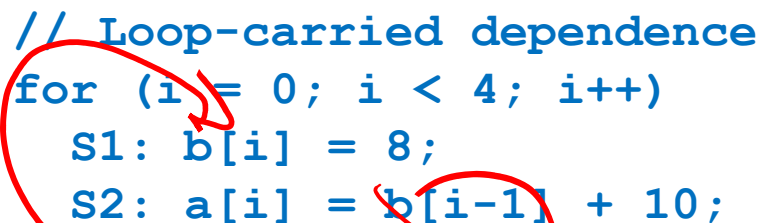
# Loop carried dependence

Loop-carried dependencies and loop independent dependencies are determined by the relationships between statements in iterations of a loop.

**Loop-carried dependence**  $\Leftrightarrow$  A statement in one iteration of a loop depends in some way on a statement in a different iteration of the same loop.

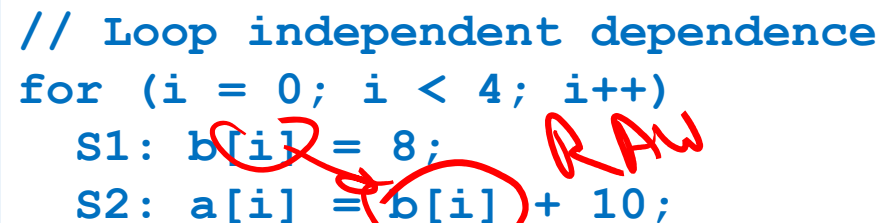
**Loop-independent dependence**  $\Leftrightarrow$  A statement in one iteration of a loop depends only on a statement in the same iteration of the loop.

```
// Loop-carried dependence
for (i = 0; i < 4; i++)
    S1: b[i] = 8;
    S2: a[i] = b[i-1] + 10;
```



*between iterations*

```
// Loop independent dependence
for (i = 0; i < 4; i++)
    S1: b[i] = 8;
    S2: a[i] = b[i] + 10;
```



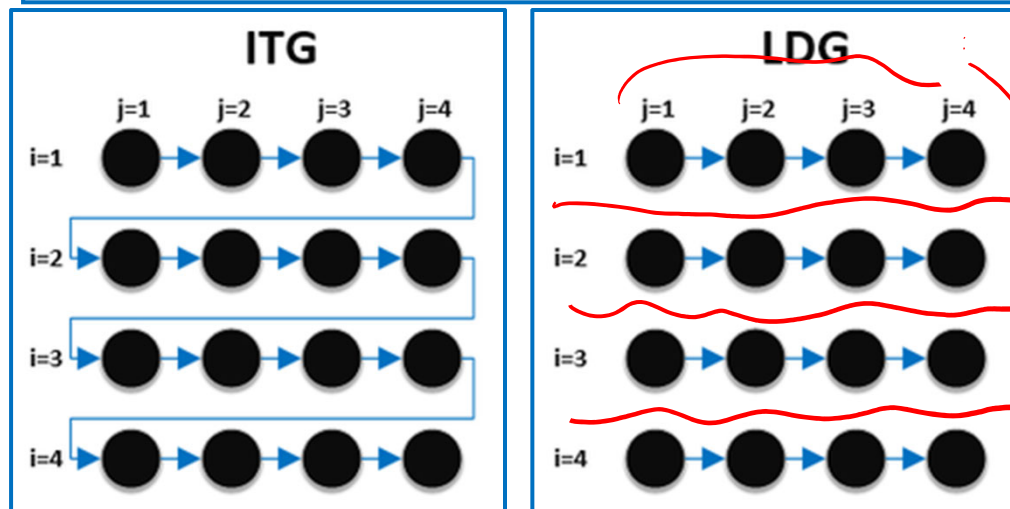
*RAW*

# Loop carried dependence, cont.

**Iteration space traversal graph (ITG)**  $\Leftrightarrow$  Shows the path that the code takes when traversing through the iterations of the loop. Each iteration is represented with a node.

**Loop carried dependence graphs (LDG)**  $\Leftrightarrow$  Gives a visual representation of all true dependencies, anti dependencies, and output dependencies that exist between different iterations in a loop. Each iteration is represented with a node.

```
// data dependence btw statements  
// in different iterations of j  
for (i = 0; i <= 4; i++)  
    for (j = 0; j <= 4; j++)  
        S1: a[i][j] = a[i][j-1] * x;
```



# Dependences and Transformations

Why is the finding of dependences so important? Because it tells us when we can transform code while maintaining correctness.

**Fundamental Theorem of Dependence:** *Any reordering that preserves every dependence in a program preserves the meaning of that program.*

**Valid Transformation**  $\equiv$  *A transformation is valid for the program to which it is applied if it preserves all dependences in the program.*

```
DO I = 1, N
  T = A(I)
  A(I) = B(I)
  B(I) = T
```

ENDDO

Vectorizable? No – Scalar T creates a bottleneck to the vector computation: only one element can be transferred at a time in the program as specified.

Parallelizable?

A&K

```
DO I = 1, N
  T(I) = A(I)
  A(I) = B(I)
  B(I) = T(I)
```

ENDDO

Vectorizable? Yes – Expand scalar temp T into a Vector temp T(1...N).

- Note: legality of transformations like these can be determined by examining the dependences in the program.
- Note: we did something like this to solve the accumulator problem in L02.

Parallelizable?



# Sometimes we can relax (a little)

**Valid Transformation**  $\equiv$  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

// If we do the interchange, do we violate a dependency?
// If we do the interchange, is the code still correct?
```

**Conclusion:** *Valid Transformations* preserve a condition that is stronger than equivalence

# Parallelization versus Vectorization

**Principal strategy** → Parallelize outer loops, vectorize inner loops (e.g., through strip mining)

Example: Inner (j) loop should definitely be vectorized. We could parallelize the outer (i,k) loops. We probably do not have  $n^2$  processors, but we can partition the outer loops (dot products) over the processors  $P$  that we have so that we execute  $N^2/P$  dot products on each processor.

Note that scalar  $r$  is OK here ( $T$  was a problem a few slides back). The difference is that  $r$  is read only in the vectorized loop, as in DAXPY.

```
/* ikj */
for (i=0; i<n; i++)          // parallelize
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)  // vectorize
            c[i][j] += r * b[k][j];
    }
```

# Extended Example: SOR

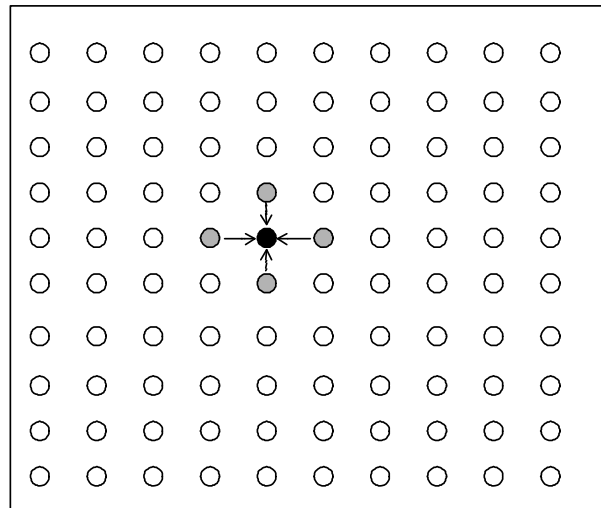
## SOR ⇔ Successive Over Relaxation

- *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

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

<i>10</i>	<i>9</i>	<i>8</i>	<i>7</i>
<i>9</i>	<i>?</i>	<i>?</i>	<i>6</i>
<i>8</i>	<i>?</i>	<i>?</i>	<i>5</i>
<i>7</i>	<i>6</i>	<i>5</i>	<i>4</i>



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j - 1] + A[i - 1, j] + A[i, j + 1] + A[i + 1, j])$$

# 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 =  $O(IN^2)$

## Serial Code

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

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

10. procedure Solve (A) /*solve the equation system*/
11.   float **A; /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.   int i, j, done = 0;
14.   float diff = 0, temp;
15.   while (!done) do /*outermost loop over sweeps*/
16.     diff = 0; /*initialize maximum difference to 0*/
17.     for i ← 1 to n do /*sweep over nonborder points of grid*/
18.       for j ← 1 to n do
19.         temp = A[i,j]; /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]); /*compute average*/
22.         diff += abs(A[i,j] - temp);
23.       end for
24.     end for
25.     if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

# Decomposition and Mapping

*→ into tasks (code)* *→ own processors (HW)*

- Idea -- whatever is independent can be run concurrently
- Try to “decompose” the computation into those independent pieces for parallel execution
- 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 – why?)

```
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
```

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 = parallelism =  $N^2$

# Look at Dependences – start with ~~ITG~~

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

Red on right – dep this iter  
 Blue on right – dep on prev. iter  
 Continue Statement  
 6

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

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

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

BLUE – Constants

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

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

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

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

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

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

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

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

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

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

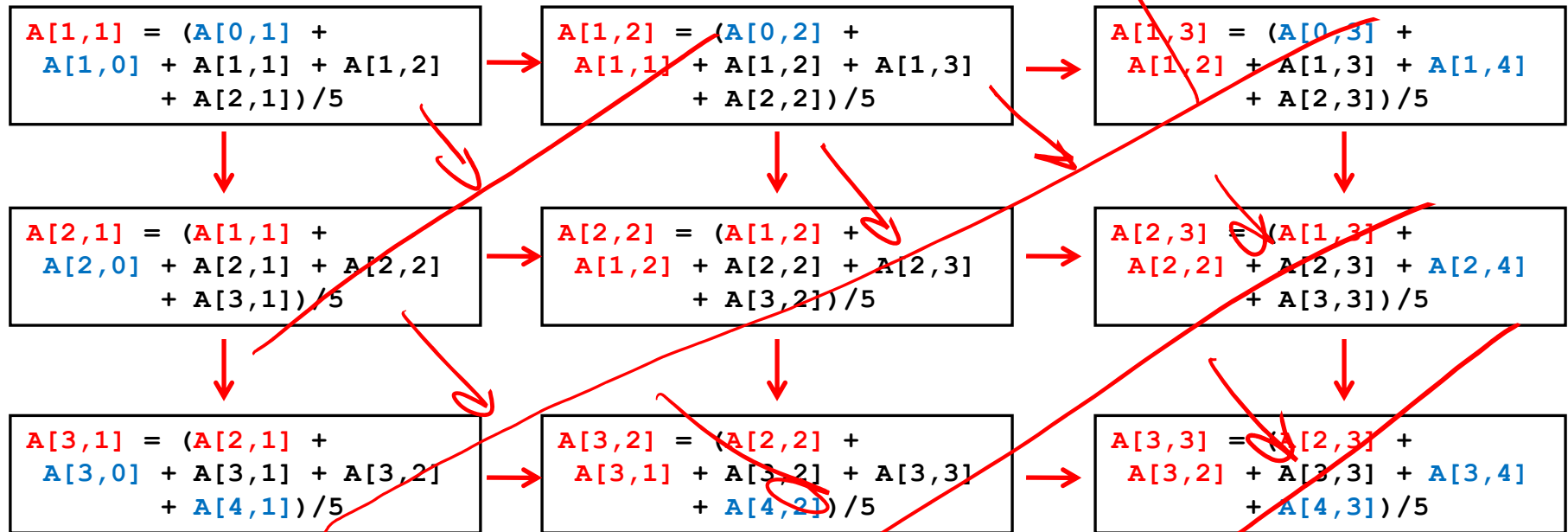
# Look at Dependences – DAG

```
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)



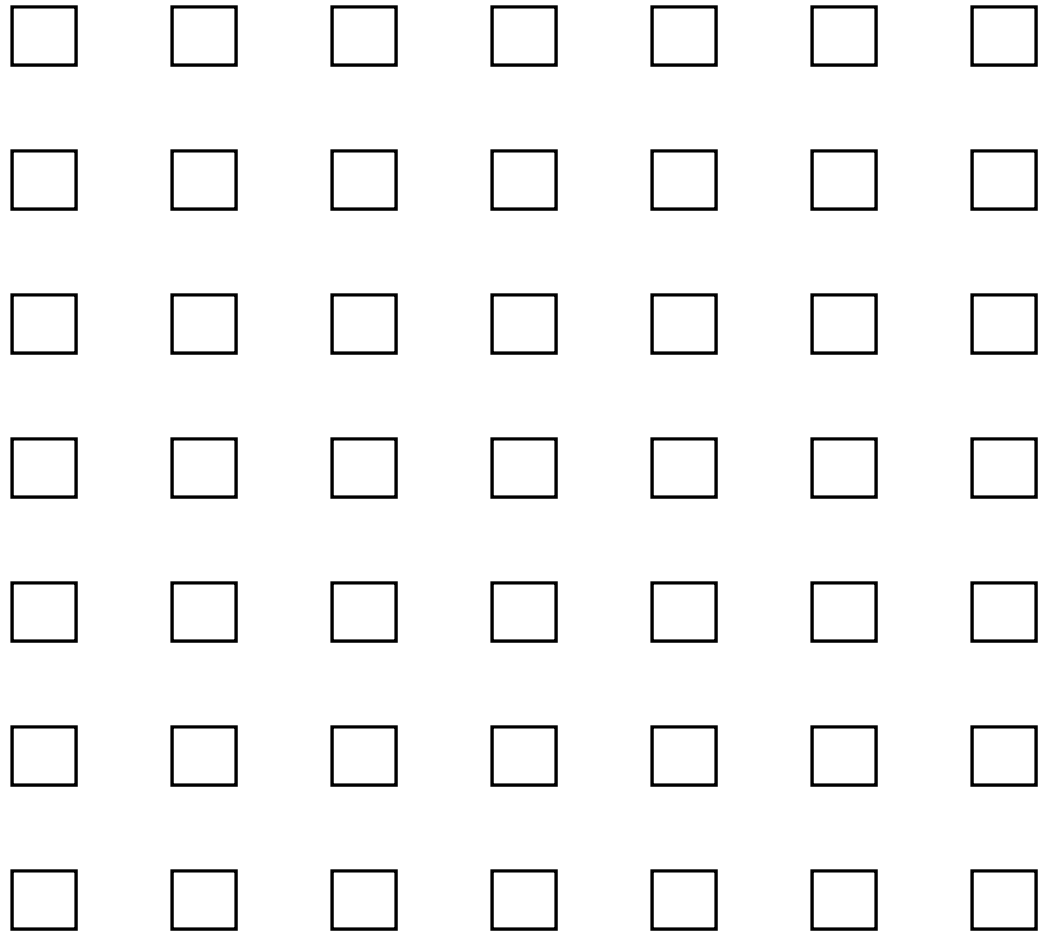
- Remove execution order
- **Arrows** show dependences among statements within L0
- A statement can execute when all of the inputs have completed



# Compute Model

```
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
```

- Each square is a processor (PE)
- One PE per array element (i,j)  
for an n x n array
- PEs can communicate with  
their four NEWS neighbors

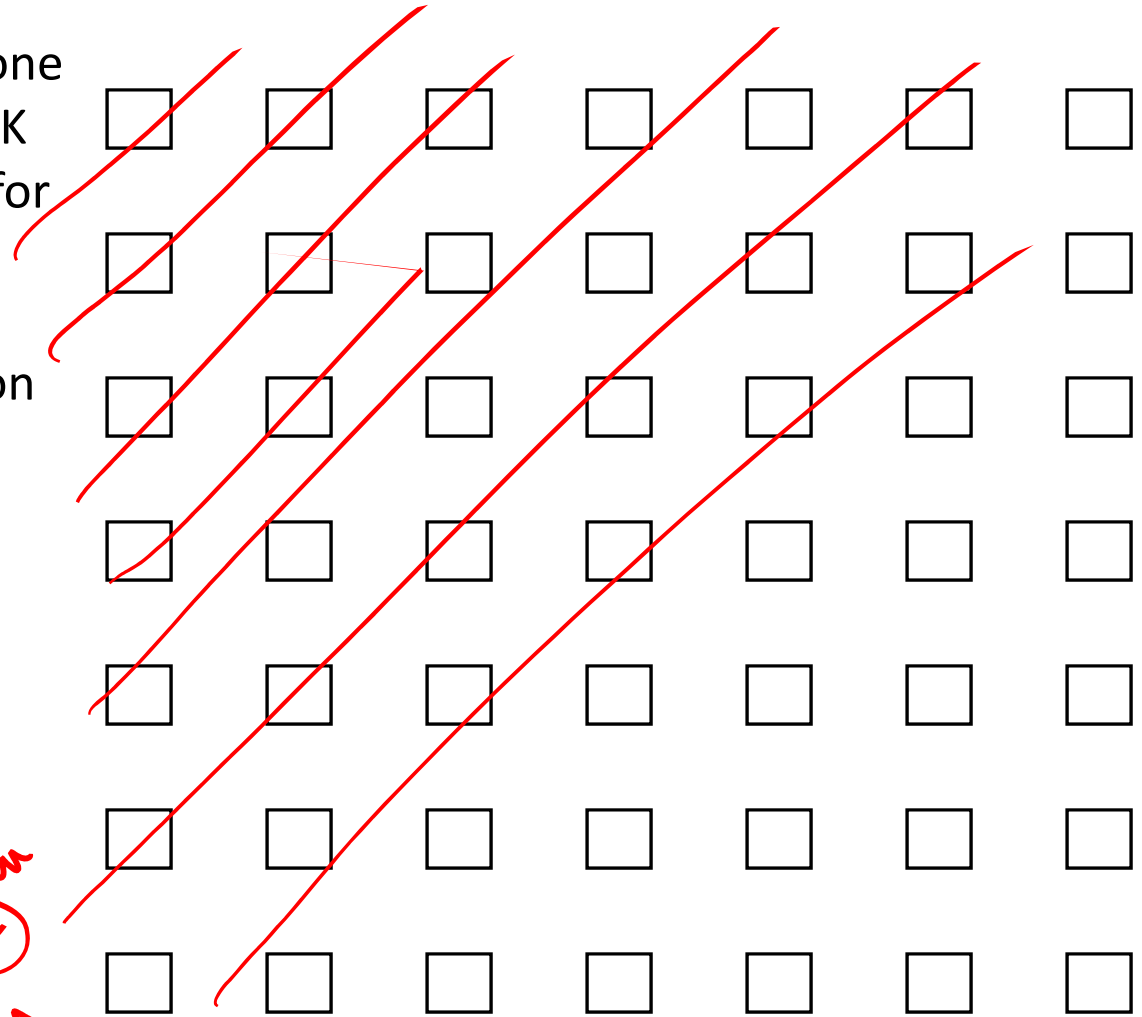


# Derive Best Execution Flow

```
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 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? —  $1/2n$



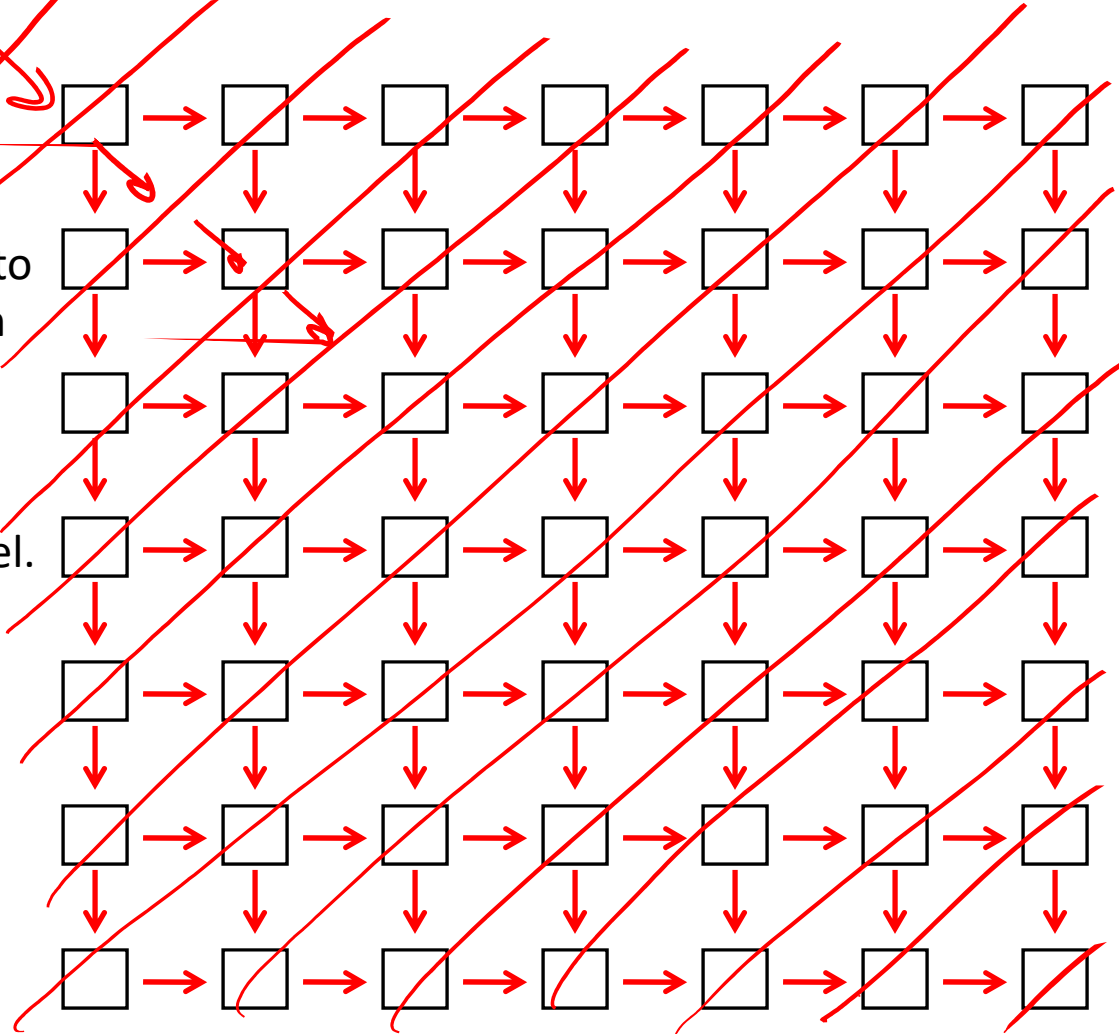
# Derive Best Execution Flow

```
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
```

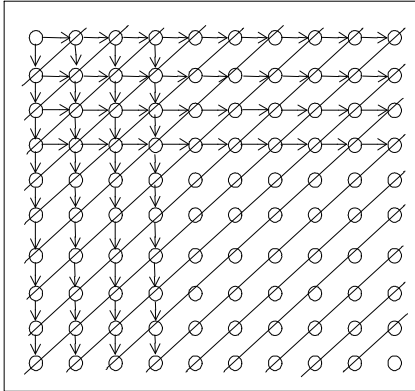
NOW → Remove L0 sync.

- 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?

*Execution is parallel*



# Parallelized Serial Code (pseudo)

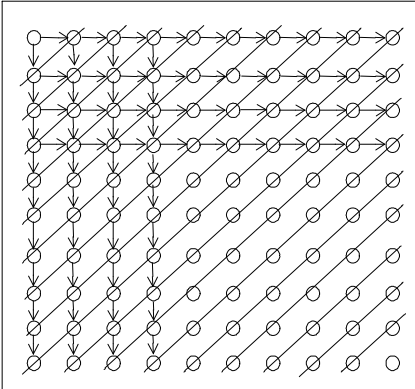


```
// Original serial code
while (not converged)
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[j,j+1] + A[i+1,j])/5
```

```
// Concurrency  $O(n)$  along antidiagonals, serialization  $O(n)$  along diagonal
// Note - synchronization is non-trivial
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**

# Remove convergence constraint



```
// Concurrency  $O(n)$  along antidiagonals, serialization  $O(n)$  along diagonal
// Note - synchronization is non-trivial
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
```

```
// Concurrency =  $O(n^2)$ .
```

```
// Note - synchronization is non-trivial
```

```
DO 1 TIMES // Assume we know how long until convergence
```

```
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
```

Full utilization? 😊

# Map to an NxN parallel processor

Start Up  
Complexity =  $O(N)$

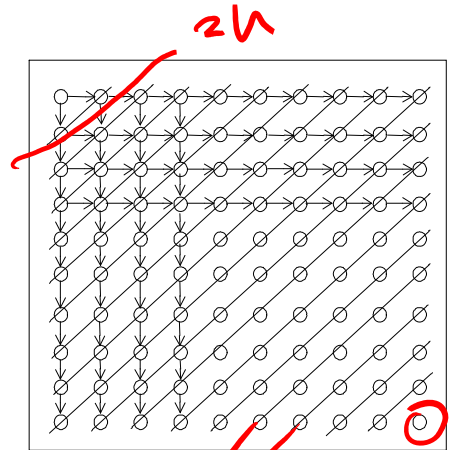
3	(3)	2	(2)	1
(3)	2	(2)	1	(1)
2	(2)	1	(1)	0
(2)	1	(1)	0	0
1	(1)	0	0	0
		(a)		

Steady State  
Complexity =  $O(1)$

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

Tear Down  
Complexity =  $O(N)$

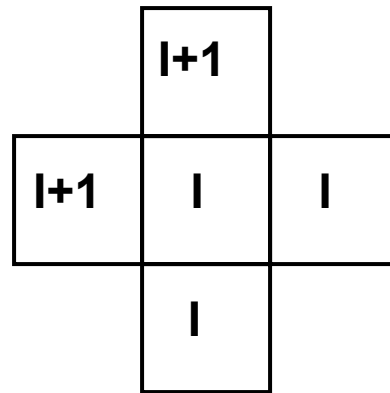
$m$	$m$	$m$	$(m)$	$m-1$
$m$	$m$	$(m)$	$m-1$	$(m-1)$
$m$	$(m)$	$m-1$	$(m-1)$	$m-2$
$(m)$	$m-1$	$(m-1)$	$m-2$	$(m-2)$
$m-1$	$(m-1)$	$m-2$	$(m-2)$	$m-3$



*2N items to get started*  
*2N items to finish*

# 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

$I$  iterations steady state phase

$2n-1$  shutdown phase

Complexity of entire computation =  $O(n) + O(I)$

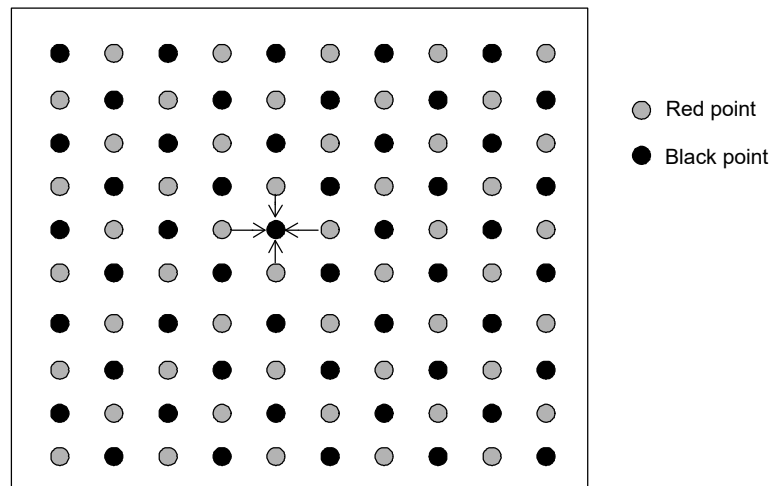
# A better-way: Red-Black decomposition

Partition array like a checker board

Red squares compute on even iteration

Black squares compute on odd iteration

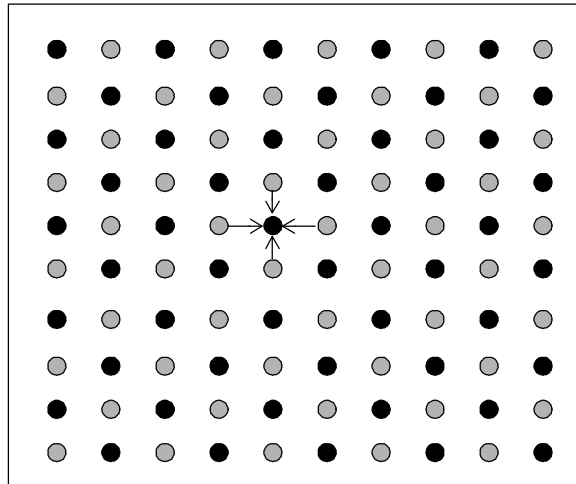
Therefore 2 cycles per iteration, but entire computation is  $O(lc)$  and independent of  $n$ .





# 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

$k+1$        $k$        $k+1$        $k$        $k+1$

$k$        $k+1$        $k$        $k+1$        $k$

$k+1$        $k$        $k+1$        $k$        $k+1$

$k$        $k+1$        $k$        $k+1$        $k$

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

$k+1$        $k$        $k+1$        $k$        $k+1$

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

.....

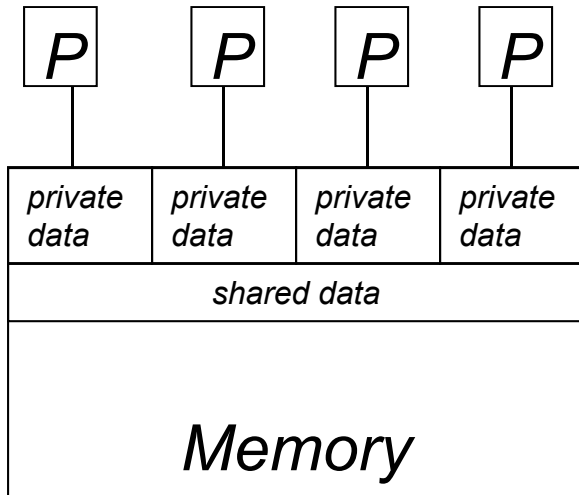
# Moral

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 Parallelizing 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

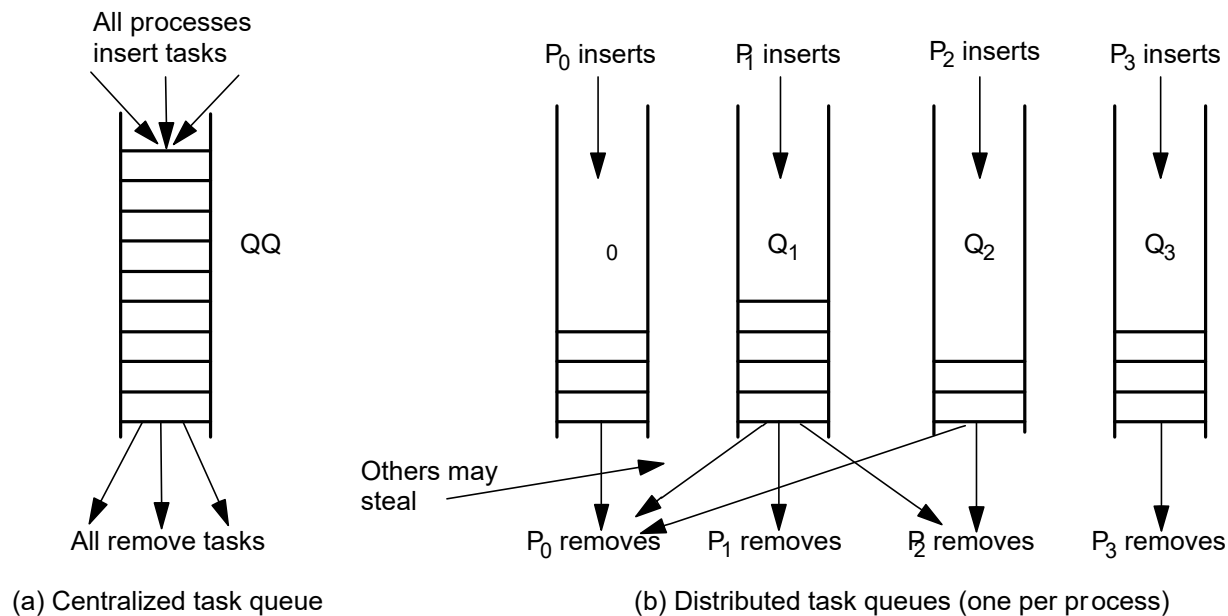
- Static:
  - Algorithmic assignment based on input; won't change
  - Low runtime overhead
  - Computation must be predictable
  - Preferable when applicable (except in multiprogrammed/heterogeneous environment)
- Dynamic:
  - Adapt at runtime to balance load
  - Can increase communication and reduce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



# 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  $\rightarrow$  computation, data access, and I/O
4. Map *processes* to *processors*

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

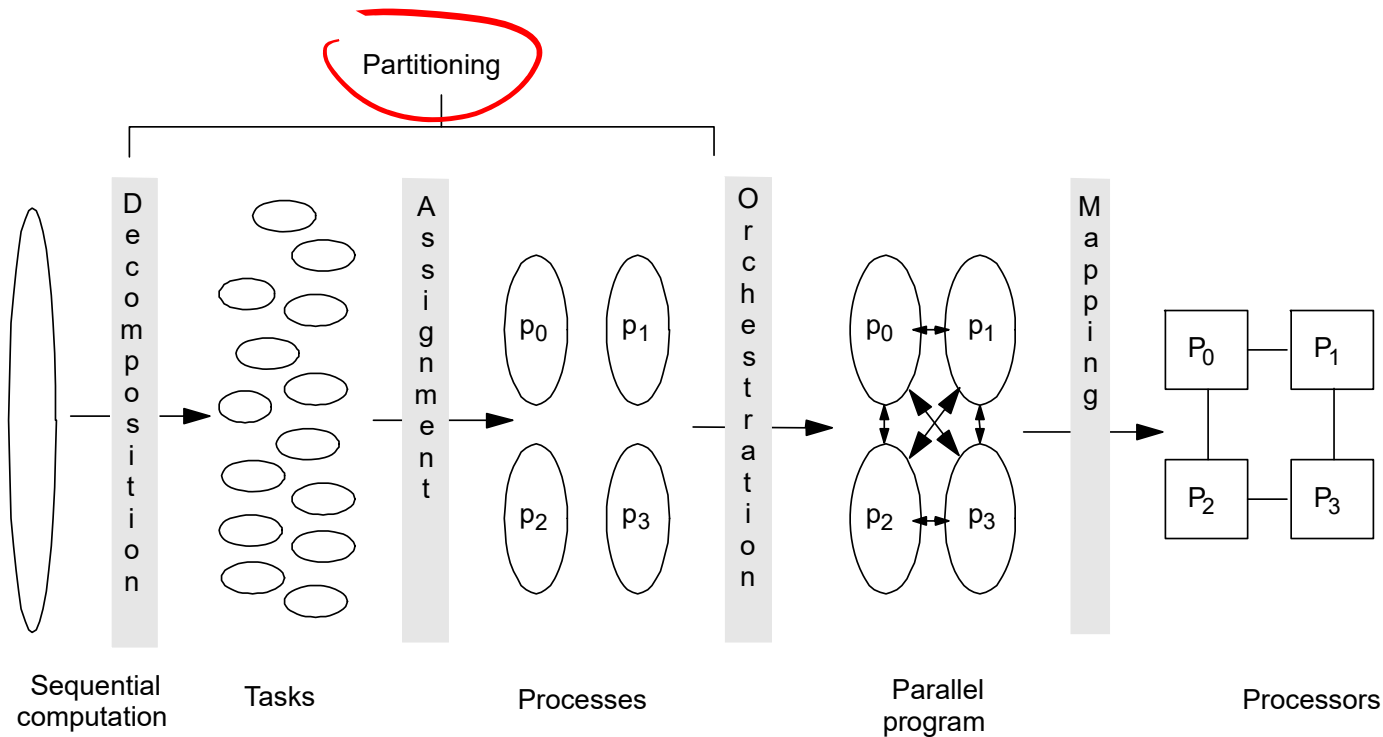
$$\text{Speedup } (p) = \frac{\text{Performance}(p)}{\text{Performance}(1)}$$



# Some Important Concepts

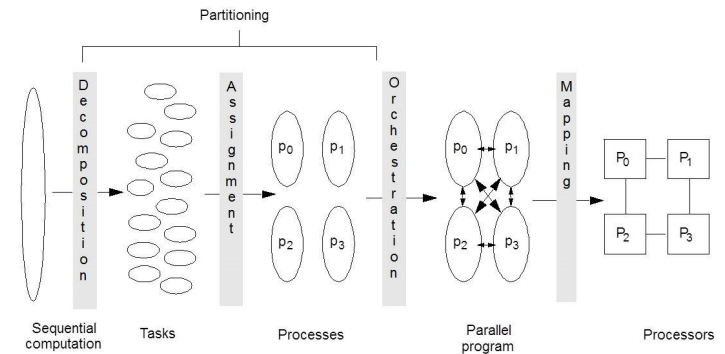
- **Task:**
  - Arbitrary piece of undecomposed work in parallel computation
  - Task instructions are executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Note different sense of *task* here and the idea of *task-level parallelism*.
    - In the sense here, we also partition dataparallel programs into tasks
- **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

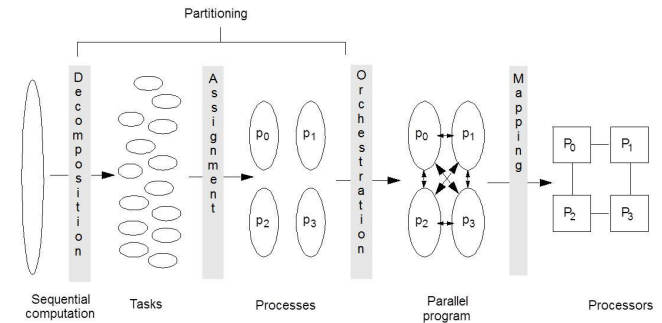


- **Decomposition**  $\Leftrightarrow$  Process of breaking up a computation into tasks (later to be divided among processes)
  - Since ...
    - *Tasks may become available statically or dynamically*
      - *Ex. Static: standard partition of a dense matrix multiply*
      - *Ex. Dynamic: event-based 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

Culler, et al.

Often requires iterative refinement

# Partitioning for Performance

Partitioning  $\Leftrightarrow$  *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.  $\rightarrow$  run all on 1 processor  $\rightarrow$  extreme load imbalance
- Maximize load balance  $\rightarrow$  random (low cost) assignment of tiny tasks  $\rightarrow$  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

Culler, et al.

# 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

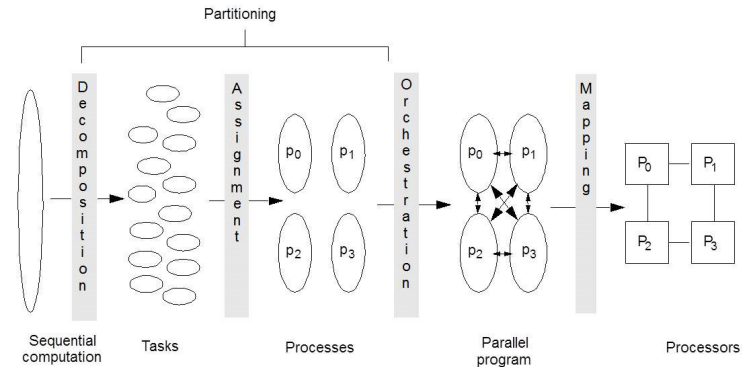
**Hard part** → Given complete knowledge of inter-task communication, solving communication and load balance is NP-hard in general case

**But in real applications** → simple heuristic solutions often work well

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

# Orchestration

**Orchestration**  $\Leftrightarrow$  How processes interact: communication, synchronization, scheduling, etc.

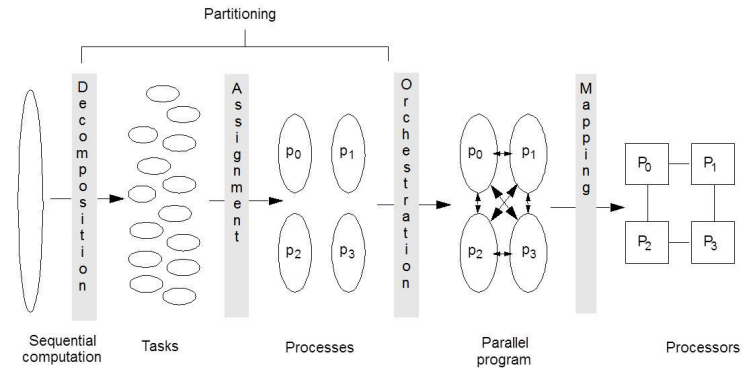


- Orchestration uses available mechanisms (e.g., hardware support, system calls, etc.) for:
    - Naming data (i.e. data access)
      - How local and remote data are accessed  $\rightarrow$  e.g., communication ops
    - Structuring communication
    - Synchronization
    - Organizing data structures 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
- Culler, et al.** More/less communication versus fewer/more processes

# Mapping

**Mapping**  $\Leftrightarrow$  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**  $\rightarrow$  Usually adopt the obvious view: process  $\Leftrightarrow$  processor (thread  $\Leftrightarrow$  core)

Culler, et al.



# *New!* Part 4: Back to SOR – full case study

Start by simplifying 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

```
15. while (!done) do                                /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do                          /*a parallel loop nest*/
18.     for for_all j ← 1 to n do
19.       temp = A[i,j];
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]);
22.       diff += abs(A[i,j] - temp);
23.     end for_all
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26. 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**, but leave the other)

**Culler, et al.** Multiple variations, see next page (also, recall parallel outer, vector inner)

Assignment

# Partitioning – Practical Parallelism w/ strips

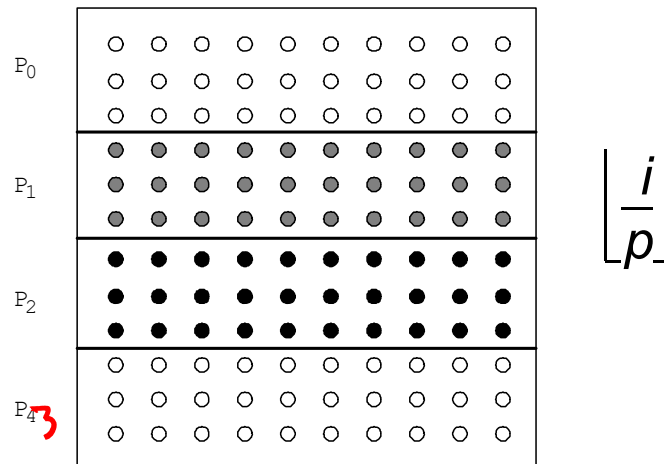
To **decompose** into rows, make **line 18** loop sequential;  $n$ -way parallel

- but implicit global synch at end of remaining **for\_all** loop

**Static assignment** by rows to processors 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

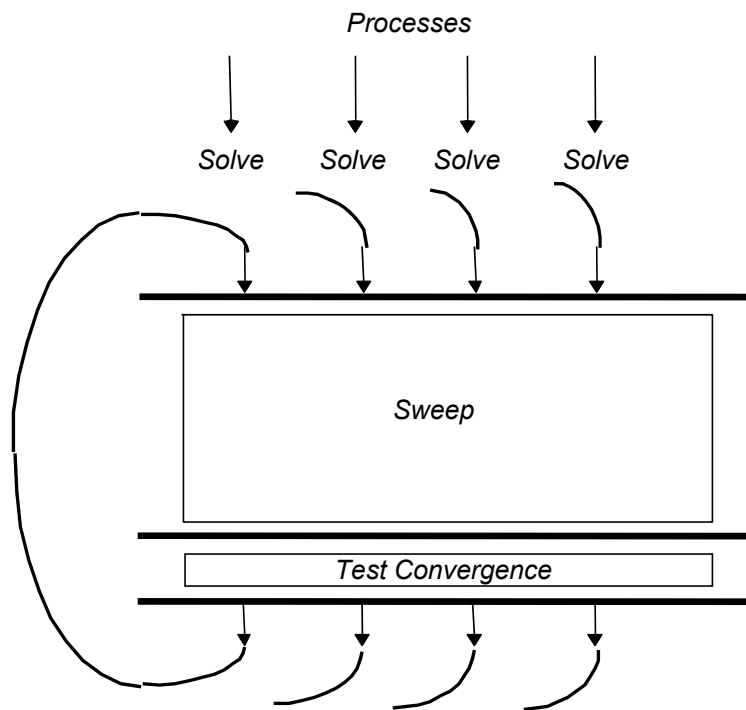


Also possible – **Dynamic assignment**: for each thread  $\rightarrow$

- get assigned a row, work on the row, get another 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

```

1.      int n, nprocs;                                /*matrix dimension and number of processors to be used*/
2.      float **A, diff;                               /*A is global (shared) array representing the grid*/
                                                    /*diff is global (shared) maximum difference in current sweep*/

2a.     LOCKDEC(diff_lock);                           /*declaration of lock to enforce mutual exclusion*/
2b.     BARDEC (bar1);                                /*barrier declaration for global synchronization between sweeps */

3.      main()
4.      begin
5.          read(n); read(nprocs);                    /*read input matrix size and number of processes*/
6.          A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.          initialize(A);                            /*initialize A in an unspecified way*/
8a.     CREATE (nprocs-1, Solve, A);                  /*create nprocs-1 worker processes*/
8.      Solve(A);                                     /*main process becomes a worker too*/
8b.     WAIT_FOR_END (nprocs-1);                      /*wait for all child processes created to terminate*/
9.      end main

10.     procedure Solve(A)
11.         float **A;                                /*A is entire n+2-by-n+2 shared array,
                                                    as in the sequential program*/
12.     begin
13.         int i,j, pid, done = 0;
14.         float temp, mydiff = 0;                   /*private variables, mydiff*/
14a.     int mymin = 1 + (pid * n/nprocs);            /*assume that n is exactly divisible by nprocs */
14b.     int mymax = mymin + n/nprocs - 1             /* and decide which rows to process*/

15.         while (!done) do                          /*outer loop until convergence*/
16.             mydiff = diff = 0;                    /*set global diff to 0 (okay for all to do it)*/
16a.         BARRIER(bar1, nprocs);                 /*ensure all reach here before anyone modifies diff*/
17.             for i ← mymin to mymax do              /*for each of myrows*/
18.                 for j ← 1 to n do                  /*for all nonborder elements in that row*/
19.                     temp = A[i,j];
20.                     A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.                     mydiff += abs(A[i,j] - temp);  /*now mydiff instead of diff*/
23.                 endfor
24.             endfor

25a.         LOCK(diff_lock);                          /*update global diff if necessary*/
25b.         diff += mydiff;
25c.         UNLOCK(diff_lock);
25d.         BARRIER(bar1, nprocs);                  /*ensure all reach here before checking if done*/
25.         if (diff/(n*n) < TOL) then done = 1;      /*check convergence; all get same answer*/
25e.         BARRIER(bar1, nprocs);
26.     endwhile

27.     end procedure

```

# Notes on SAS Program

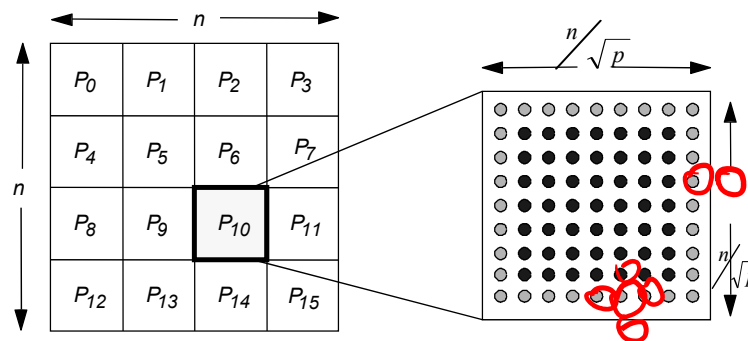
**SAS** ⇔ Shared Address Space

- SPMD: not lockstep or even necessarily same instructions
- Assignment of work to threads is 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**



$$\text{Comm} \propto \frac{4n}{\sqrt{p}}$$

$$\text{Comp} \propto \frac{n^2}{p}$$

$$\frac{\text{Comm}}{\text{Comp}} = \frac{4\sqrt{p}}{n}$$

Strong Scaling  
n fixed  
p ↑

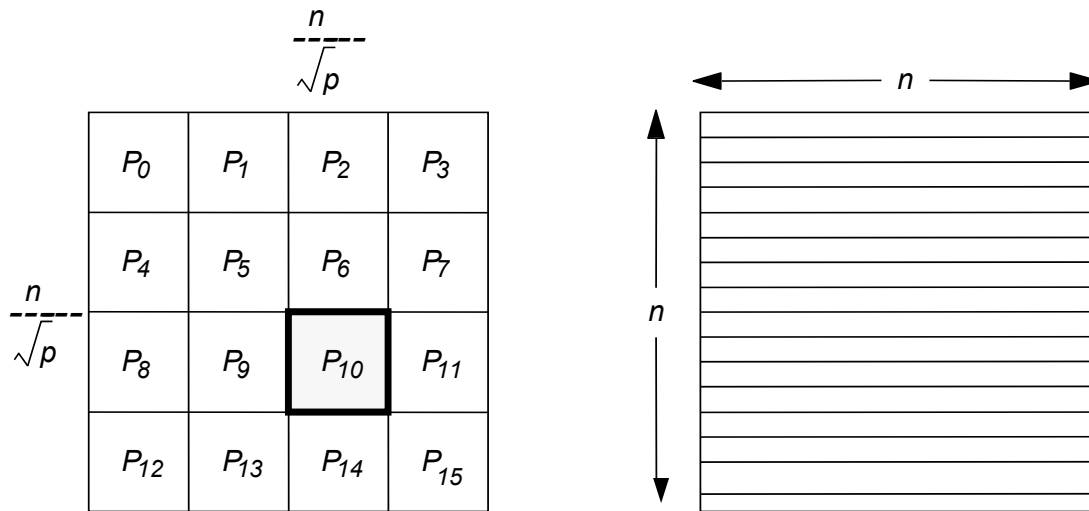
Perimeter to Area comm-to-comp ratio (in 3D: area to volume)

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

Culler, et al.

# Domain Decomposition (cont.)

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



$$\text{Comm} = 2n$$

$$\text{Comp} = \frac{n^2}{p}$$

$$\frac{\text{Comm}}{\text{Comp}} = \frac{2p}{n}$$

For both decompositions: computation per processor =  $n^2/p$

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

– Why?

– Application dependent: strip may be better in other cases

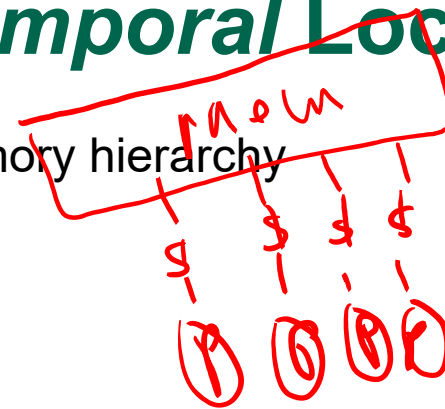
- E.g. particle flow in tunnel

Culler, et al.

Lower is better  
square  $\leq$  strip  
 $\frac{4\sqrt{p}}{n} \leq \frac{2\sqrt{p}}{n}$   
 $p > 4$  block is better  
 $4 \leq p$

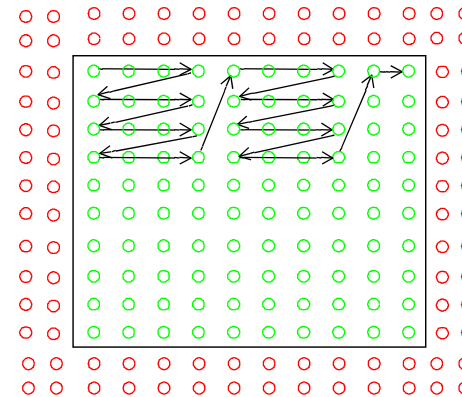
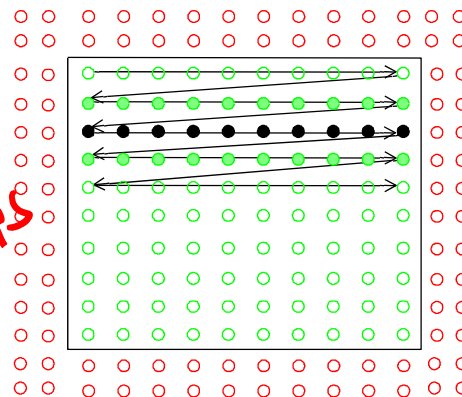
# Orchestration: Exploiting *Temporal* Locality

- 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?*



*guarantee  
5x5 sweeps  
- entire square  
fits into cache*

*Maximum  
Reuse?  
- 5  
- # of sweeps*



(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*

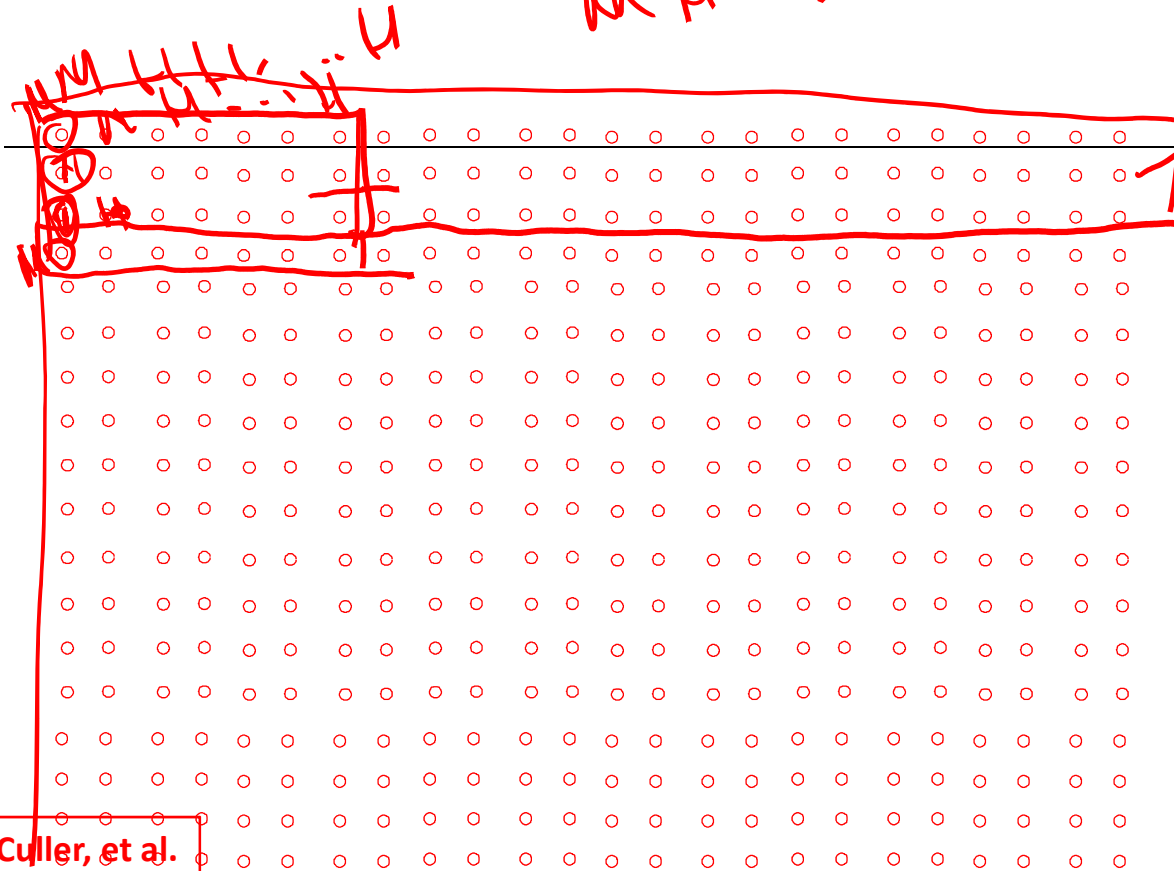


## Exploiting *Temporal* Locality, cont.

Assume that 3 rows fit in cache

Miss Rate =  $\frac{1}{8}$

B = 8 element



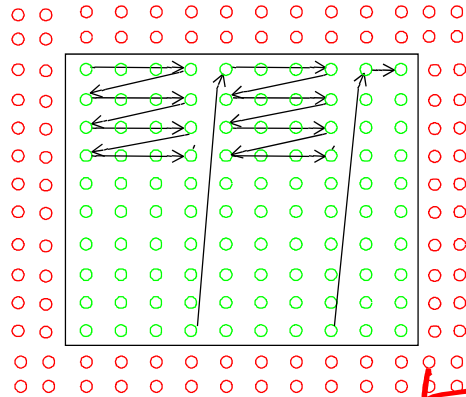
at end  
of row.  
3 rows in  
cache

2nd row - 2 rows  
all links

1 row  $\frac{1}{8}$

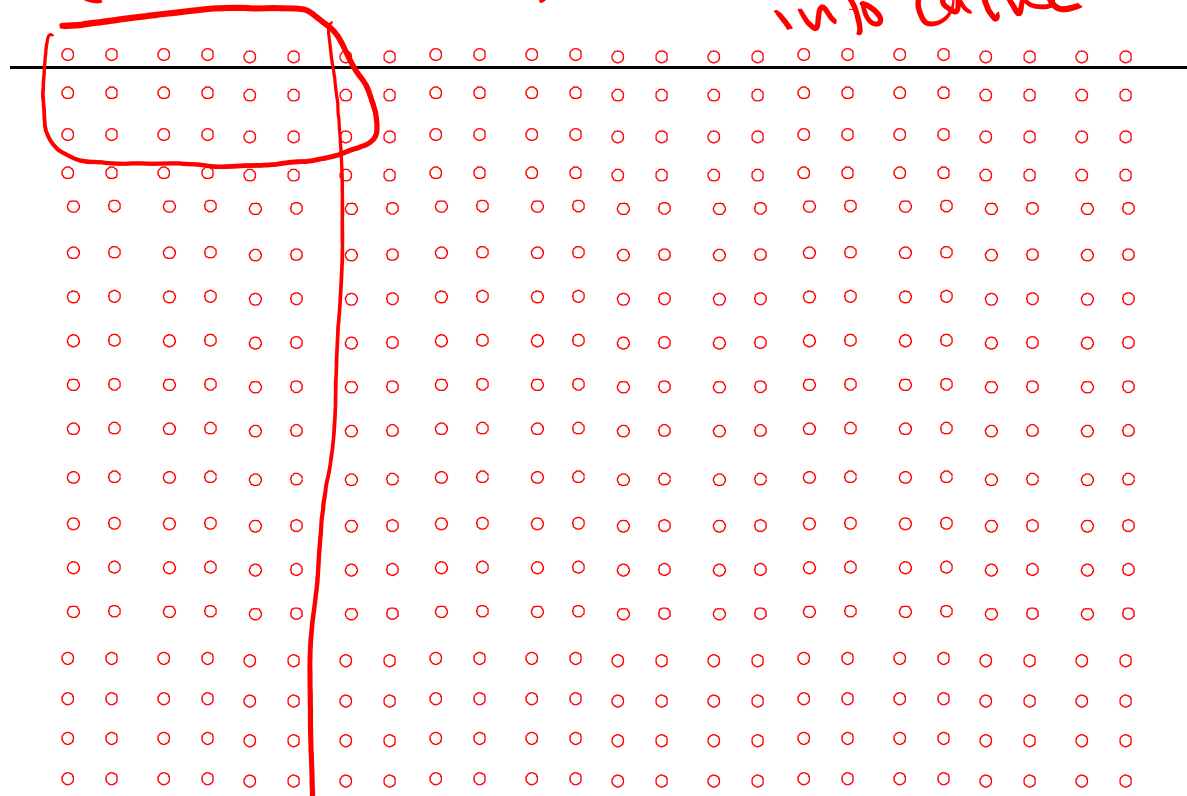
Miss rate =  $1/24$

# Exploiting *Temporal* Locality, cont.



Assume that  $< 1$  row fits in cache

choose size  
so that  
3 rows fit  
into cache



# Orchestration: Exploiting *Spatial* Locality

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

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

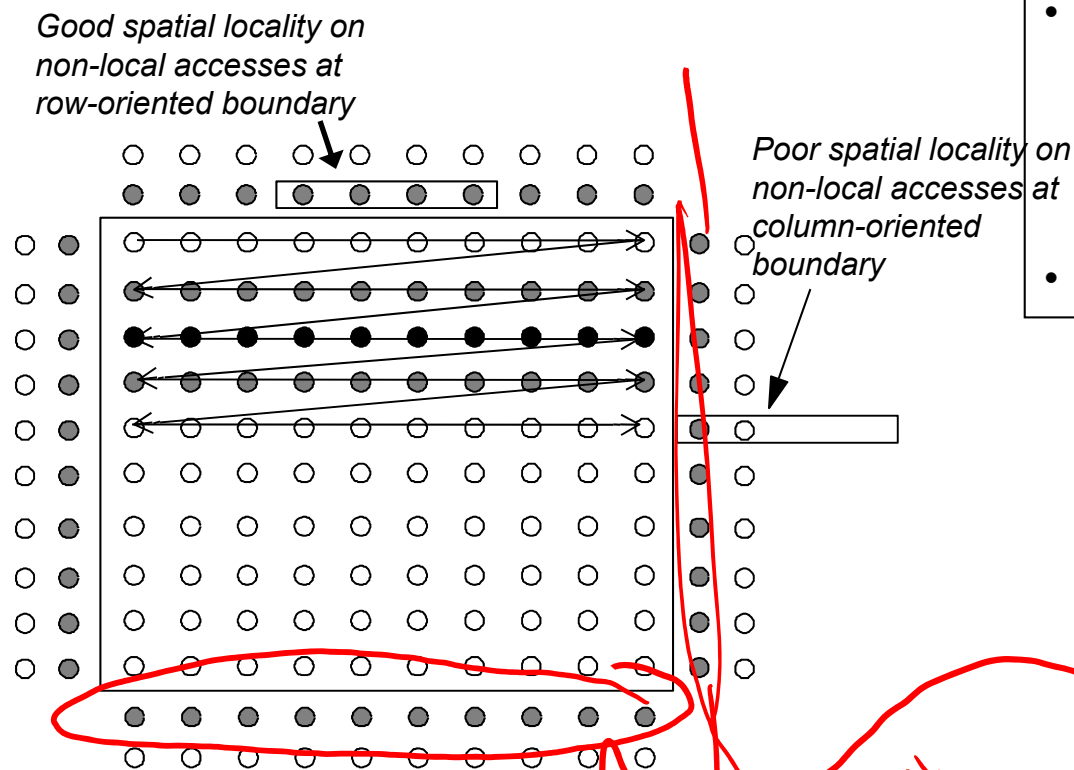
- Major spatial-related causes of artifactual communication:
  - Conflict misses
  - Data distribution/layout (allocation granularity)
  - Fragmentation (communication granularity)
  - False sharing of data (coherence granularity)
- All depend on how spatial access patterns interact with data structures
  - Fix problems by modifying data structures, or layout/alignment

# 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

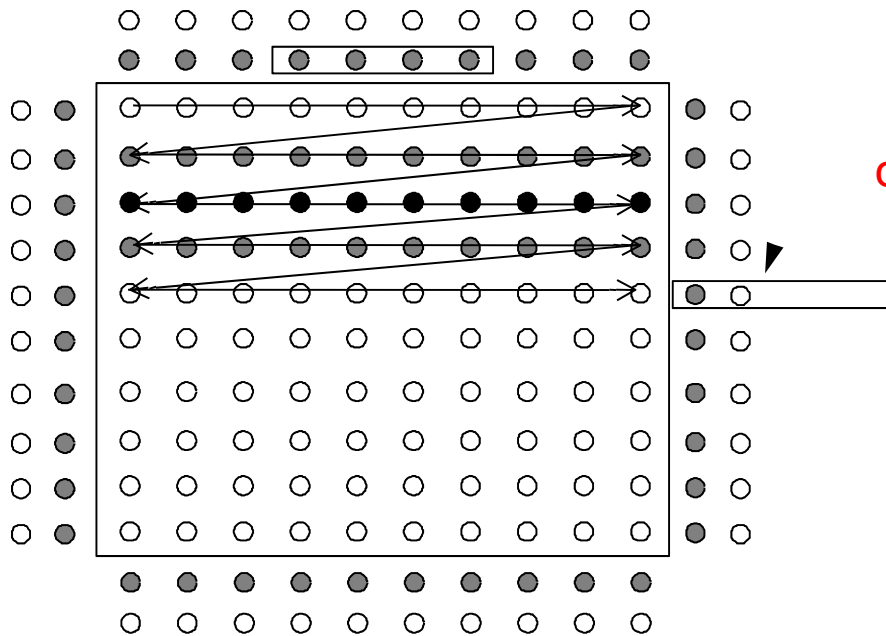
- neighboring Blocks in cache of other processors
- Write by another processor invalidates your copy (it's now stale)
- No need to worry about cache coherence mechanism: suffice that on every iteration, the first read to a cache block outside your own Block will always be a miss.
- What's better, Blocks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



Result depends on  $p$

Culler, et al.

# Tradeoffs with Inherent Communication, cont.



Total Misses = core misses  
+ top&bottom misses  
+ side misses

$$\text{core misses} = n^2/8p$$

Block

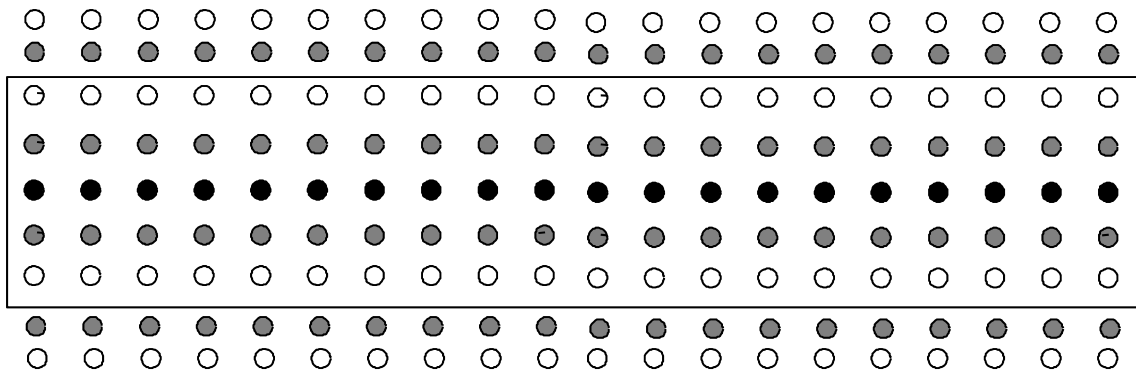
$$\text{t\&b misses} = 2n / 8\sqrt{P}$$

$$\text{side misses} = 2n / \sqrt{P}$$

Strip

$$\text{t\&b misses} = 2n / 8$$

$$\text{side misses} = 0$$



Compare non-core misses

$$\text{Strip} = n/4$$

$$\text{Block} = 9n/4\sqrt{P}$$

Break Even

$$P = 81$$

*Squares are better for  $P > 81$*

# Spatial Locality Example

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

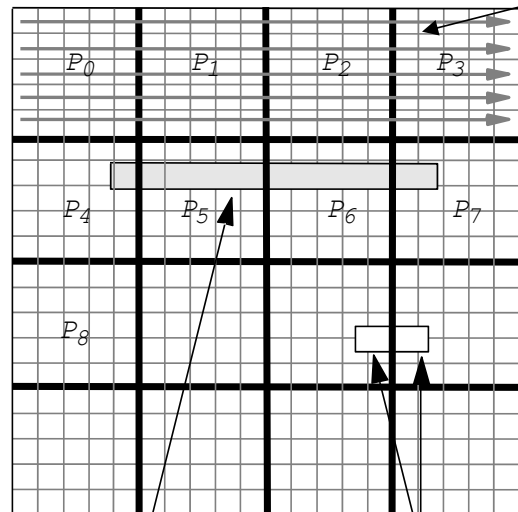
- Natural 2-d versus higher-dimensional array representation

## 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  
 $1K \times 1K \times 8B = 8 \text{ MB} = 2K \text{ pages}$   
128 pages per processor

double  
`X[1024][1024];`

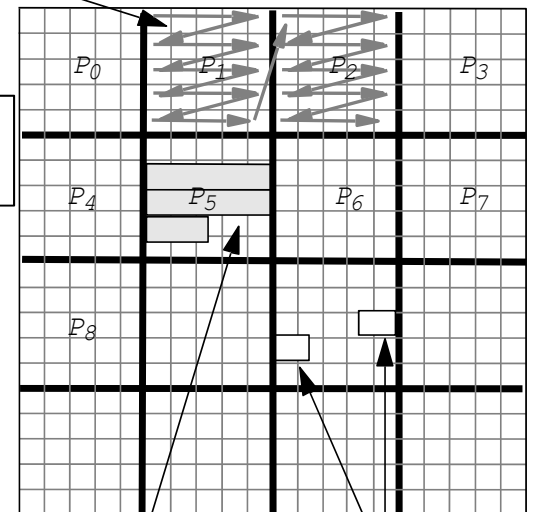


Page straddles  
partition boundaries:  
difficult to distribute  
memory well

Cache block  
straddles partition  
boundary

(a) Two-dimensional array

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



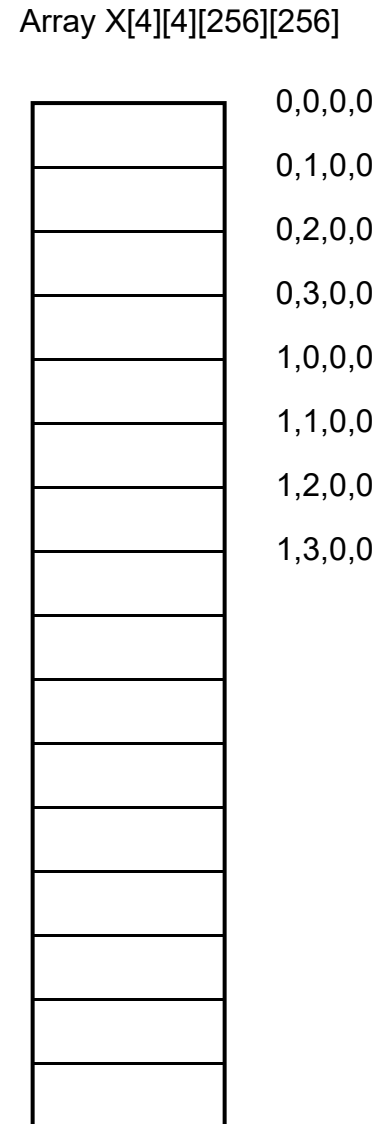
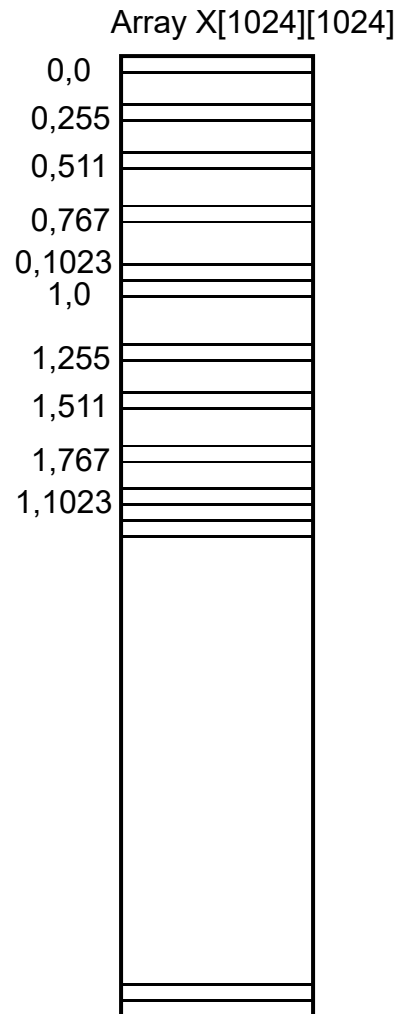
Page does  
not straddle  
partition  
boundary

Cache block is  
within a partition

(b) Four-dimensional array

Contiguity in memory layout

# Spatial Locality Example, cont.

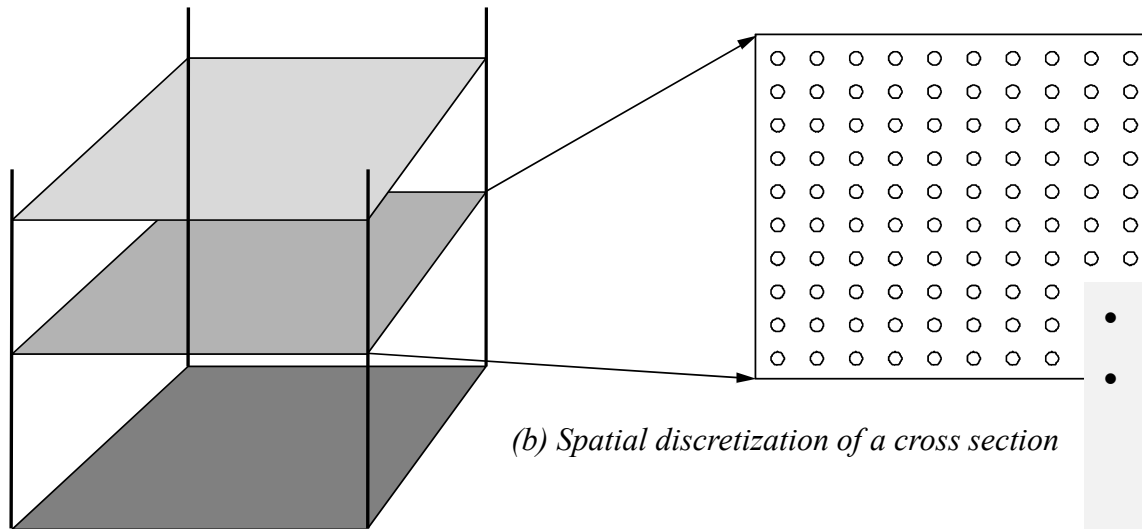


# Motivating Problems

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

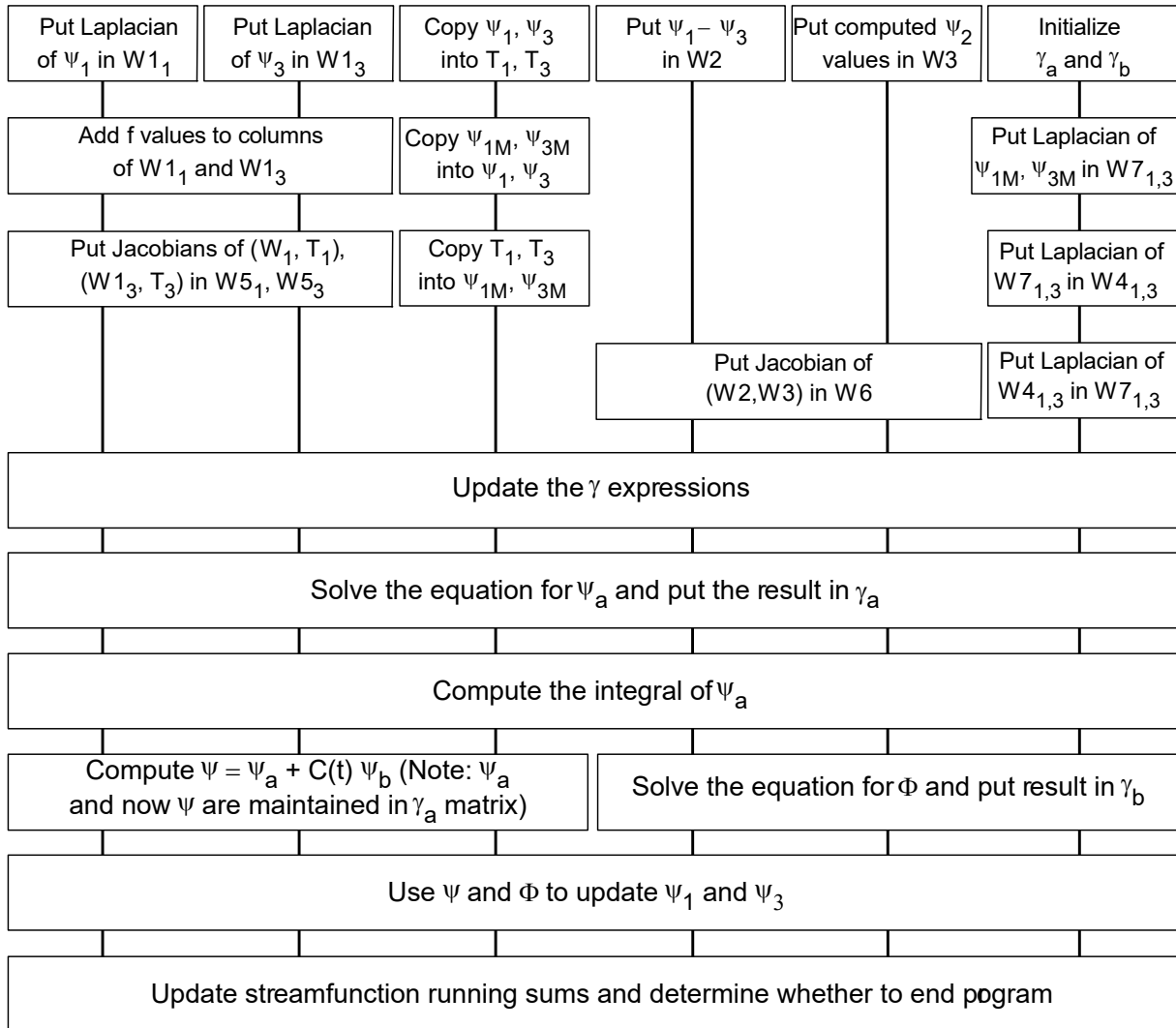
- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations
- (mostly) Local data exchange

Culler, et al.

- 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:  $n \times n$ ,  $n/2 \times n/2$ ,  $n/4 \times n/4$ , 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:



# Partitioning

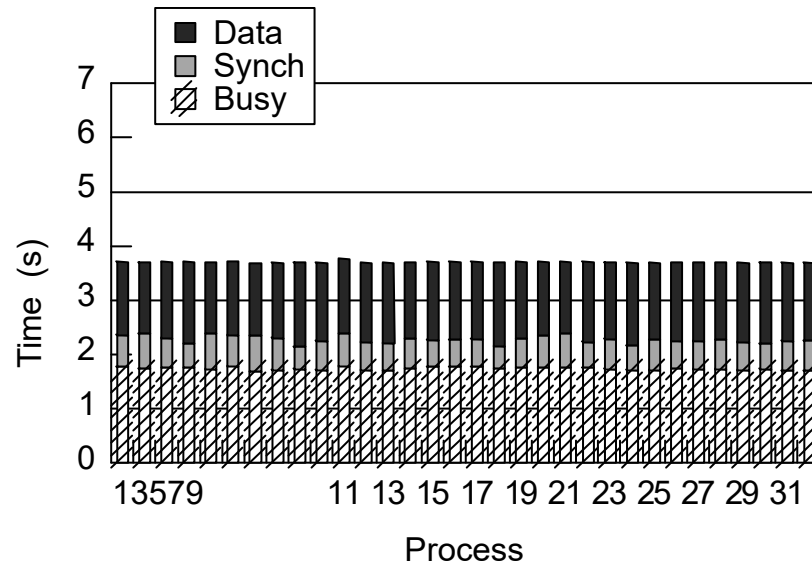
- Exploit data parallelism only
  - Function parallelism only to reduce synchronization
- Static partitioning within a grid computation
  - Block versus strip
    - inherent communication versus spatial locality in communication
  - Load imbalance due to border elements and number of boundaries
- Solver has greater overheads than other computations

# Orchestration and Mapping

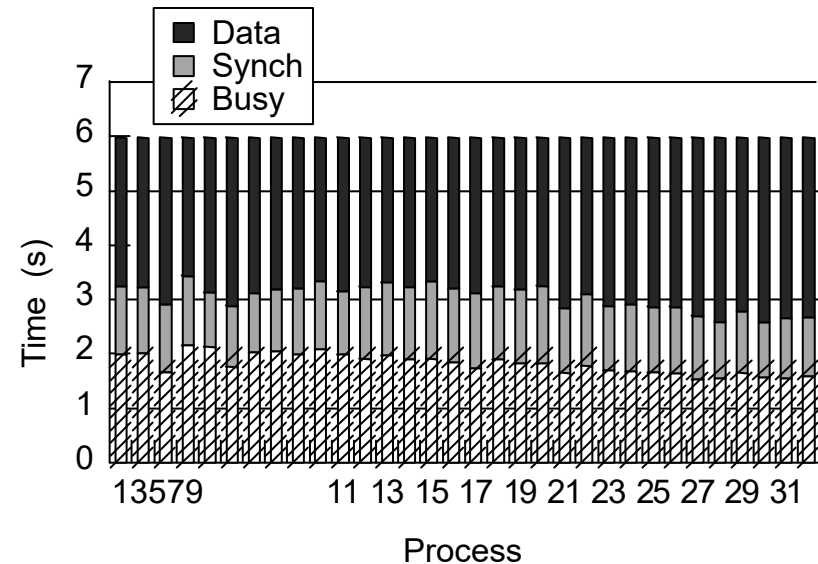
- Spatial Locality similar to equation solver
  - Except lots of grids, so cache conflicts across grids
- Complex working set hierarchy
  - (i) A few points for near-neighbor reuse, (ii) three sub-rows, (iii) partition of one grid, (iv) partitions of multiple grids in a hierarchy, (v) some data across 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**

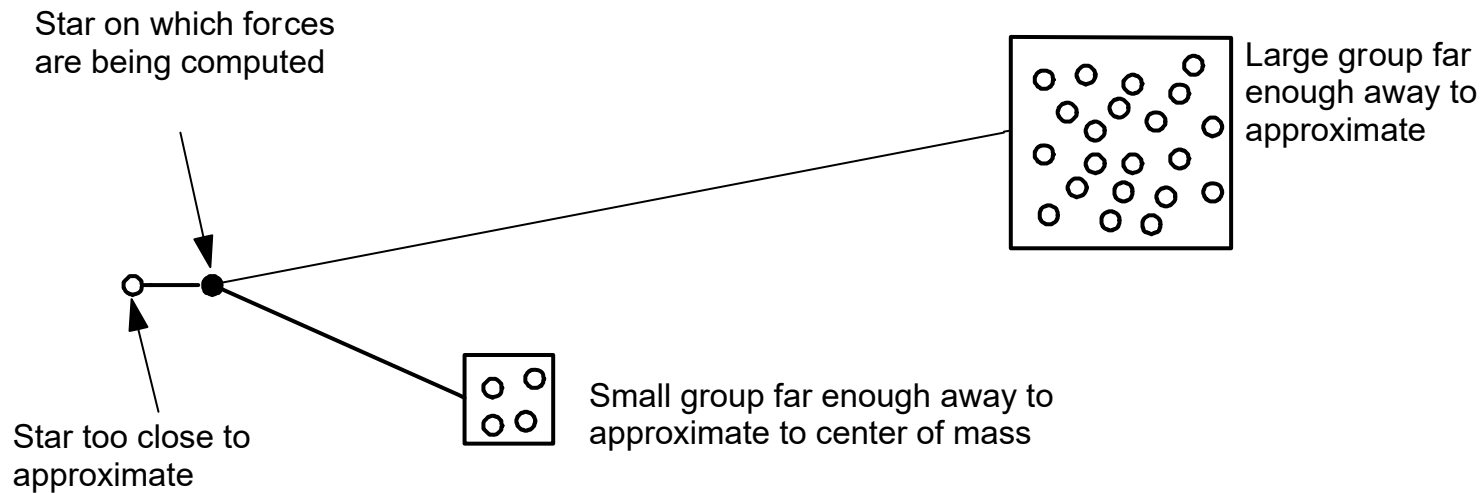


**(b) 2-d grids**

- 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

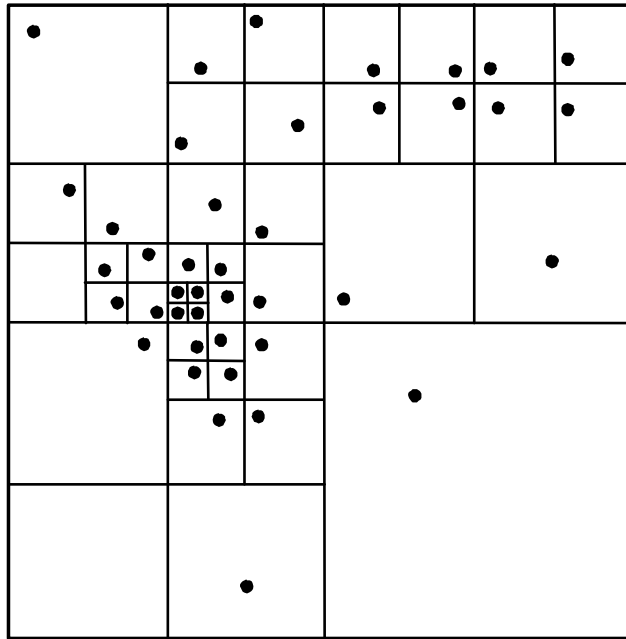
# Simulating Galaxy Evolution

- 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:  $G = \frac{m_1 m_2}{r^2}$

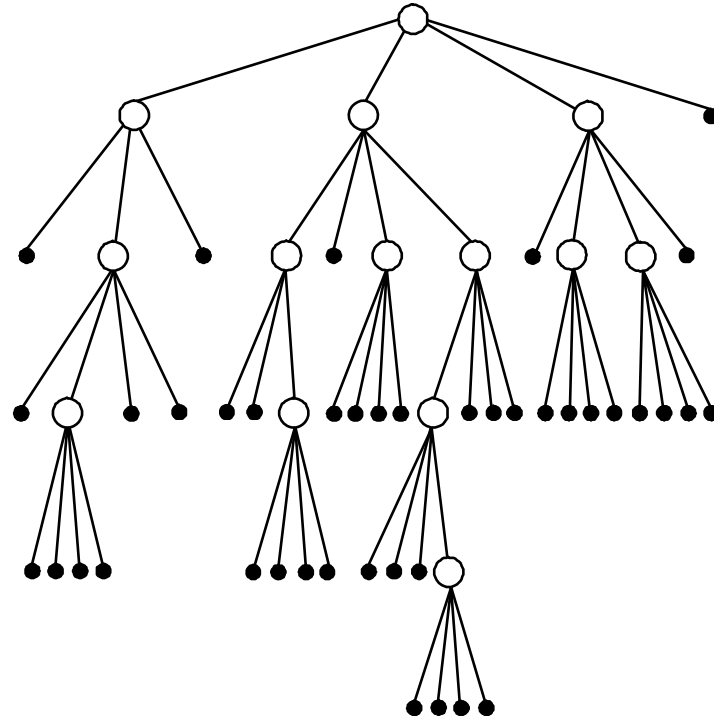


- *Many time-steps, plenty of concurrency across stars*
- *Non-uniform data structures*

## Case Study 2: Barnes-Hut



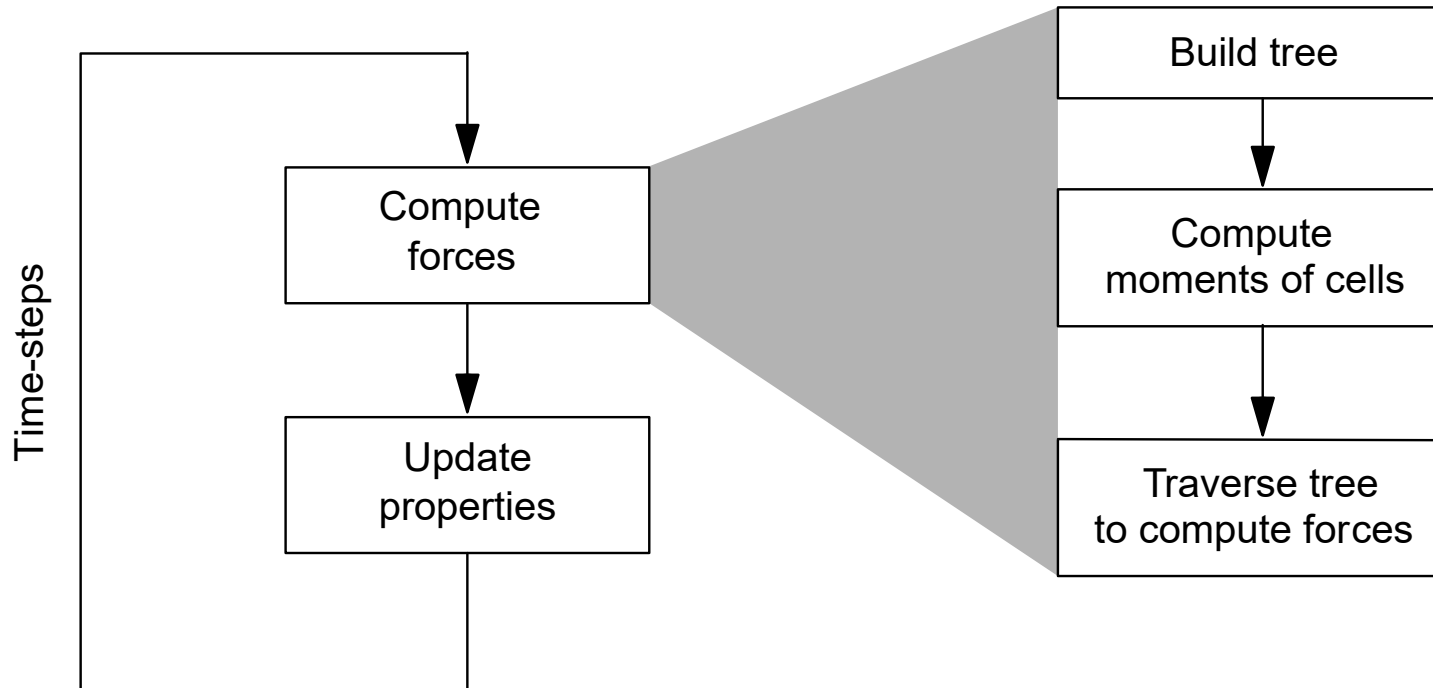
(a) The spatial domain



(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

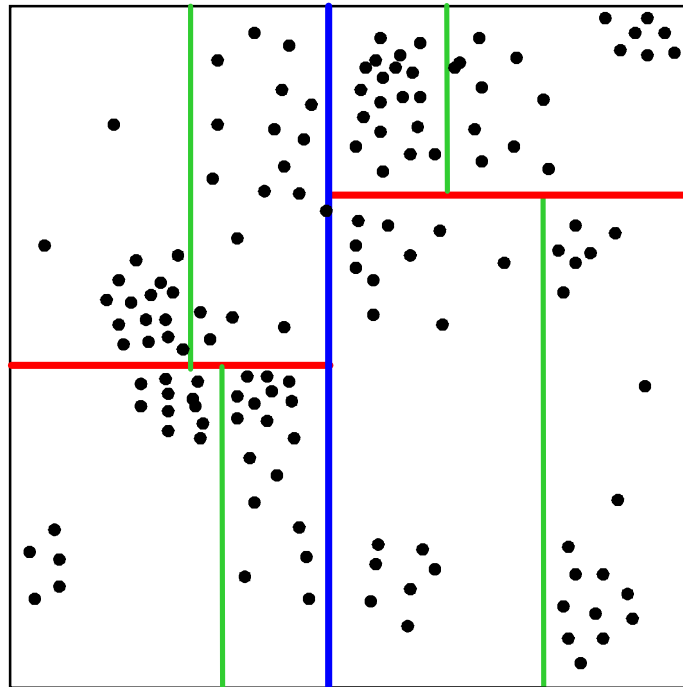
- Decomposition: bodies in most phases, cells in computing moments
- Challenges for assignment:
  - Nonuniform body distribution => work and comm. Nonuniform
    - Cannot assign by inspection
  - Distribution changes dynamically across time-steps
    - Cannot assign statically
  - Information needs fall 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  $\neq$  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

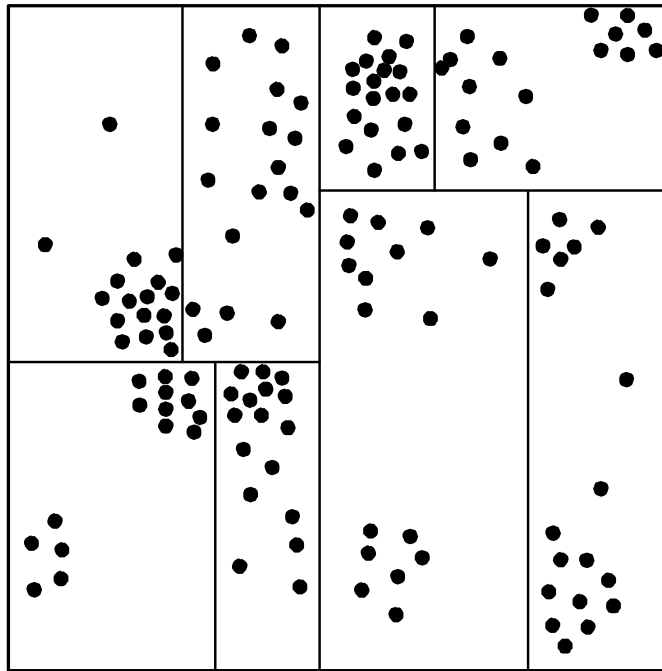
- Orthogonal Recursive Bisection:
  - Recursively bisect space into subspaces with equal work
    - Work is associated with bodies, as before
  - Continue until one partition per processor



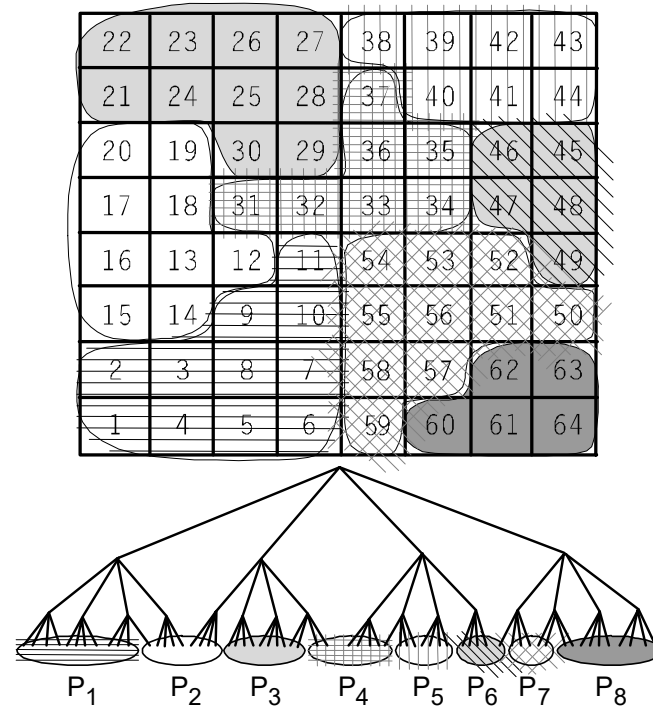
- *High overhead for large no. of processors*

# Another Approach: Costzones

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



(a) ORB

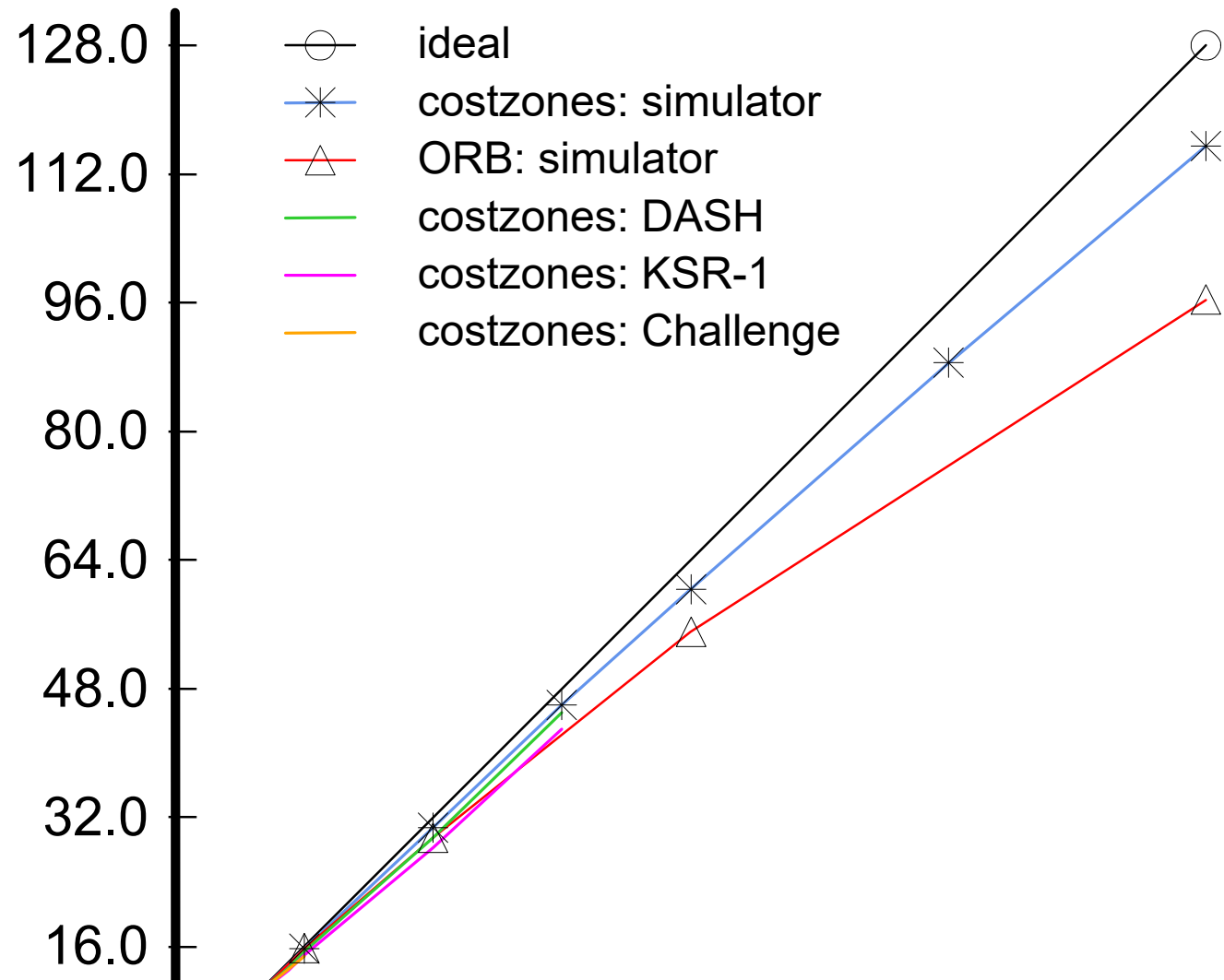


(b) Costzones

- 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



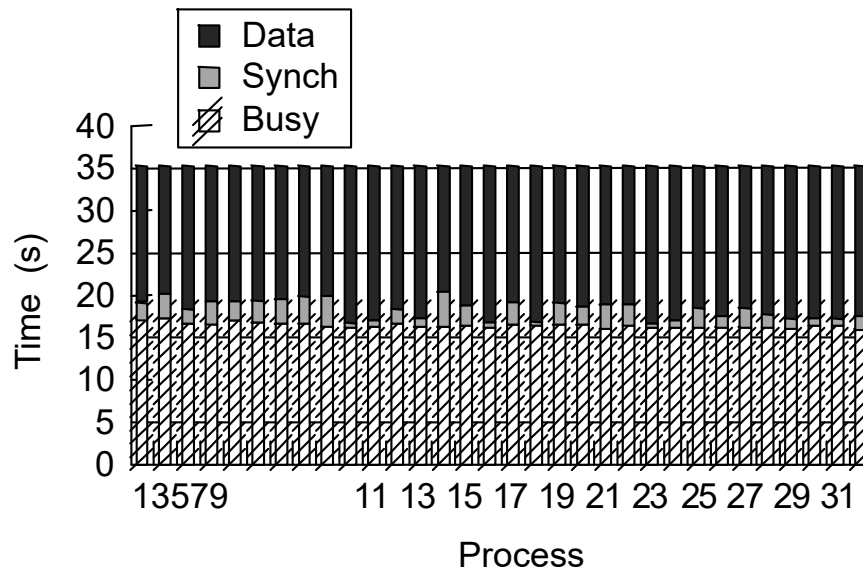
# Orchestration and Mapping

- 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

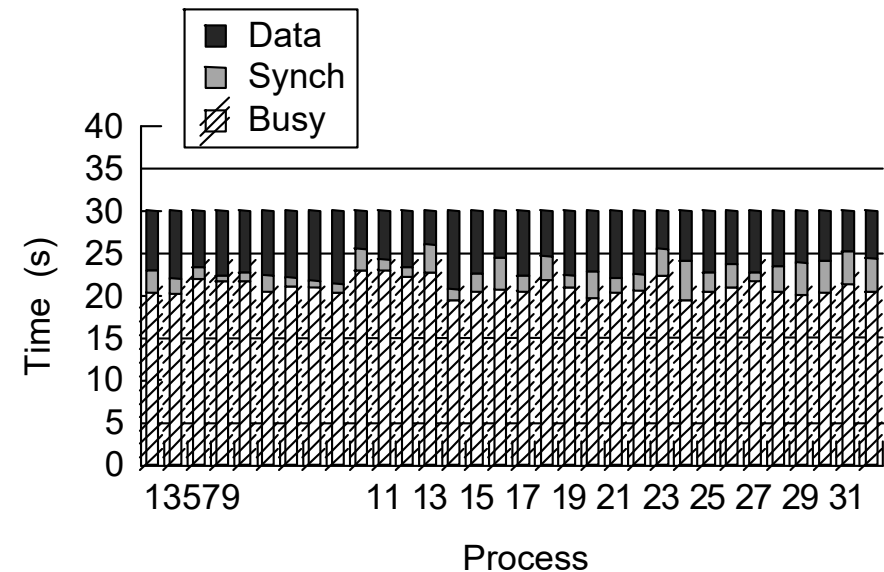
# Execution Time Breakdown

- 512K bodies on 32-processor Origin2000

- Static, quite randomized in space, assignment of bodies versus 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

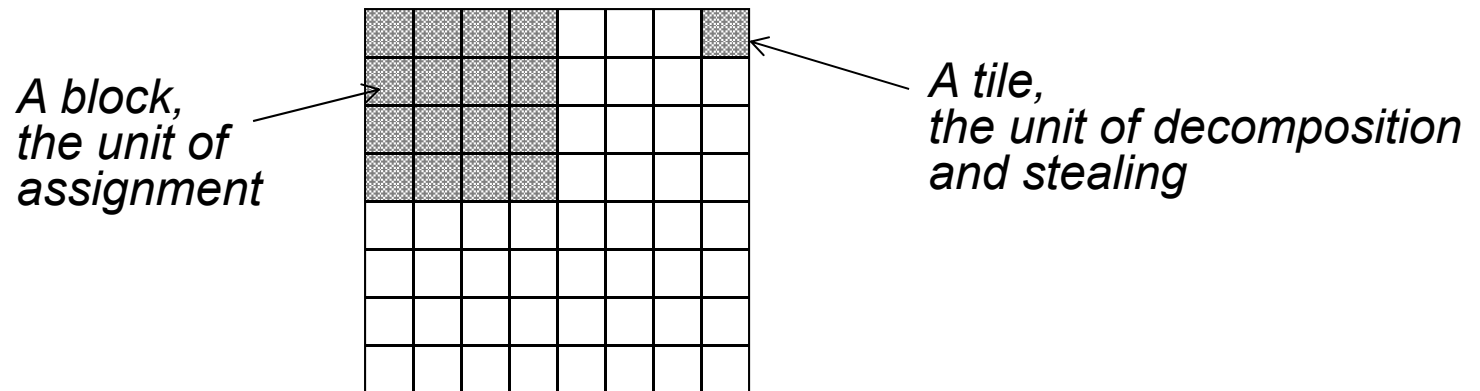


# Raytrace

- 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

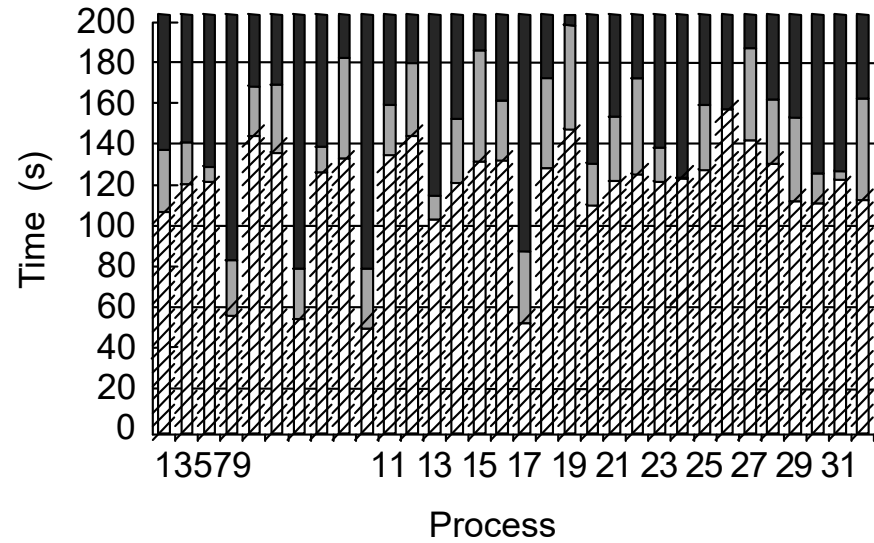
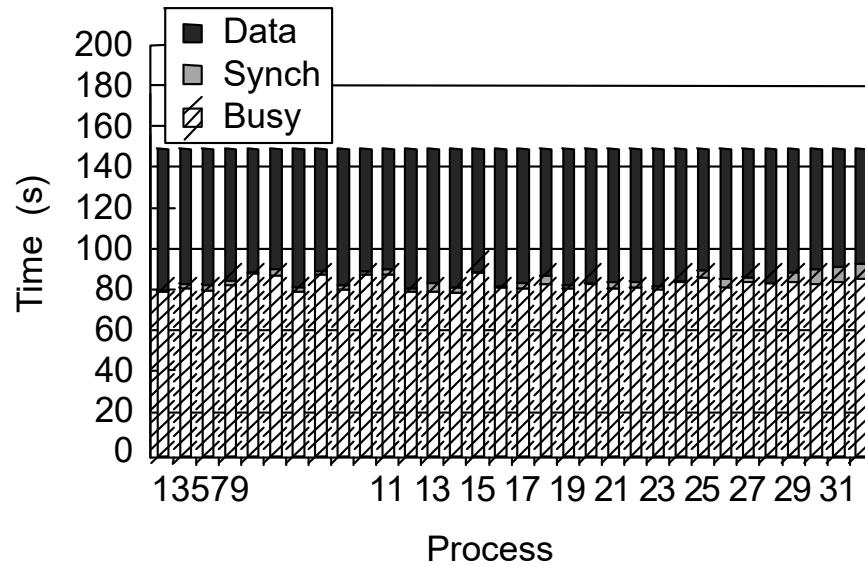


*Could use 2-D interleaved (scatter) assignment of tiles instead*

# 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