CS315A/EE382B: Lecture 4

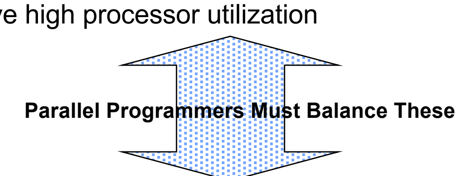
Application Parallelization II: Mapping & Data

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

The Two Sides of Parallelization

- **Dividing Work:** Need to chop computation into parallel tasks
  - What are smallest independent units in a program?
  - Achieve high processor utilization

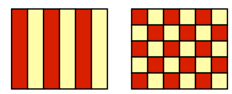


- **Partitioning Data:** Localizing data onto processors
  - Must *map* tasks to processors along with data
  - Necessary on message passing machines
  - Very helpful on shared-memory machines
  - Minimize expensive interprocessor *communication*

Review: Static Partitioning with OpenMP & pthreads

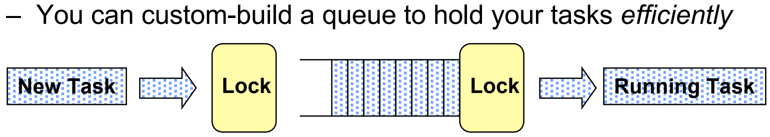
- Do it manually with pthreads
  - Choose how to pass iterations to threads

- OpenMP offers simple options for loops
  - `schedule(static, size)` distributes *size* iterations/CPU
    - Simple and clear
    - Nesting works in some environments
      - Works under Solaris 10
      - Usually use entire rows/columns of multi-D arrays
    - Can get stuck if you (# *iterations*)/(*size•n_procs*) not an integer
      - Some "extra" processors during last batch of blocks
  - This covers most common cases



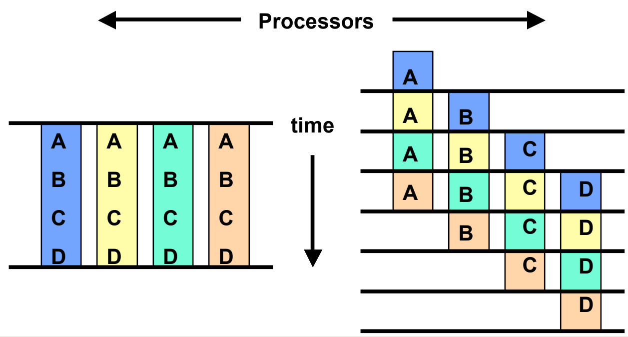
Review: Solution: Dynamic Partitioning

- Use *real* threads (pthreads) for **large** parallel tasks
  - *Examples:* Entire database queries, web page lookups
  - Let the underlying thread system handle scheduling
    - Pthreads includes many routines to control scheduling
    - Saves you a lot of work
    - Allows *pre-emption* of long running tasks
- Use hand-built *task queues* for smaller parallel tasks
  - *Examples:* Tree nodes, blocks of pixels, etc.
  - Avoids often overly general thread schedule model
  - You can custom-build a queue to hold your tasks *efficiently*



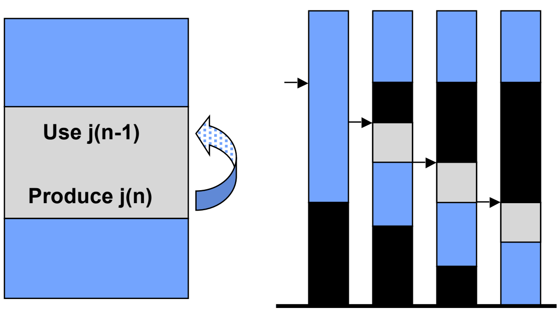
Review: Pipeline Parallelism

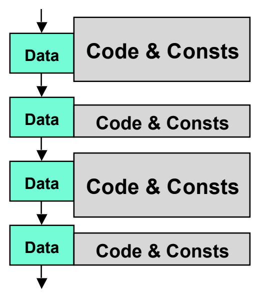
- There are two common ways to parallelize:
  - Execute same task on different processors
    - Processor executes whole task
  - Pipeline task across processors
    - Processor executes a piece of a task



Review: When to Use Pipelining

- Loop carried dependencies are the main reason:





- Can also sometimes be more natural with streaming data
  - Data arrives over time
  - As new data arrives, old moves on through
  - Code, constants & loop carried information doesn't move!

Today's Outline: Mapping & Data Management

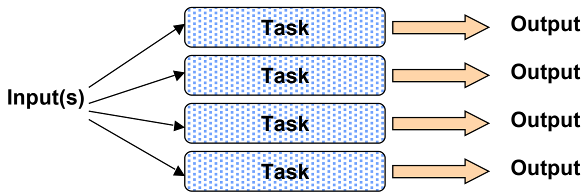
- Basic task mapping
  - Find overall application dependency graphs
  - Divide into dependent phases of tasks
  - Map in time (between phases) and space (within phases)
- *Mapping for Regular Applications:* Array blocking techniques
  - Control how we break regular structures into tasks
  - Similar concepts can be applied to irregular structures
    - But beyond the scope of what can be covered in-class
- *Reductions:* Reducing data dimensionality
- Steps to parallelize a full application!

Quick Review: Tasks

- Tasks are the smallest parallel code regions in a program
  - We've looked at using them, individually
- We have discussed two characteristics:
  - Runtime (constant between tasks or varying?)
  - Number (predictable?)
  - These affect task scheduling (static/dynamic)
- Now we must look at another characteristric: **Data Use**
  - Input-output to/from each task sets communication
  - Provides opportunities for parallelism
  - Affects sequencing & processor mapping

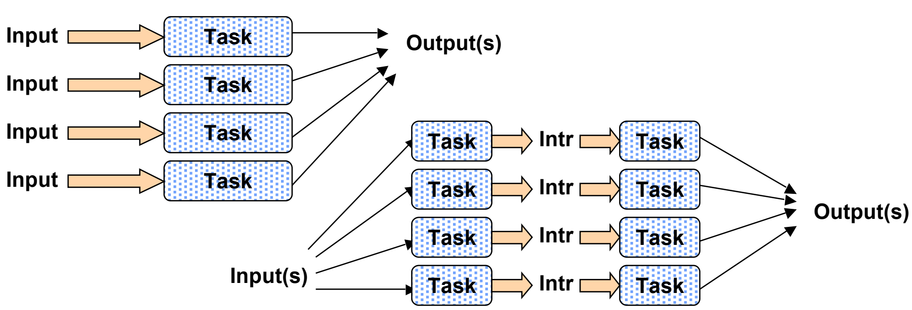
Task Breakdown by Data I

- Data parallelism is most prevalent sort of parallelism
- Domain decompositon
  - Divide data into pieces and then associate tasks with pieces . . . But how?
- 1) Independent output data produced by independent tasks
  - Minimizes data *written* from one processor to another
  - Use it first, if possible



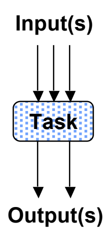
Task Breakdown by Data II

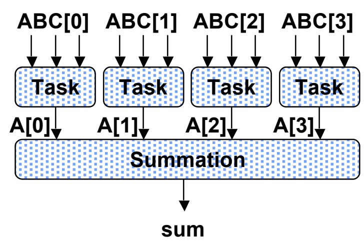
- 2) Allocating a task per input data
  - Simple and straightforward to find
  - Must often communicate elsewhere for output (e.g. sum, max)
- 3) Known intermediate values offer mid-program “I/O” points
  - Offer a sort of “hybrid” technique



Task Flow Graphs

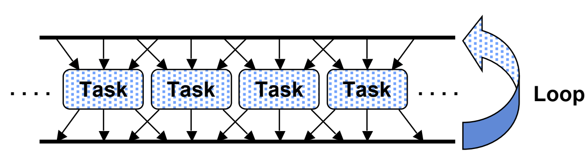
- We can represent a parallel application as a graph of tasks
  - Directed, with data output(s)-to-input(s) of next task(s)
  - Flows down from the program start to end





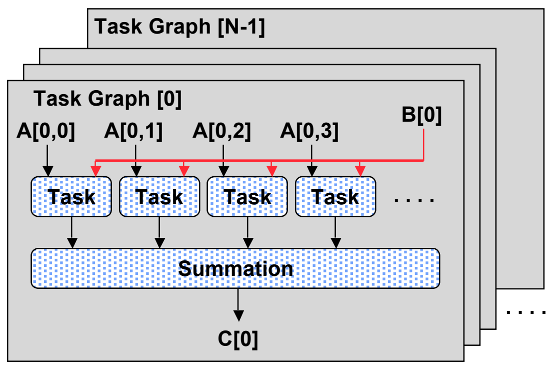
"Real" Task Graphs I

- Real applications are much more complex
  - Often easier to represent some things as loops
  - Often easier to group many-to-many communication
  - *Example 1:* Physics timestep simulation



"Real" Task Graphs II

- *Example 2:* Matrix-vector multiplication

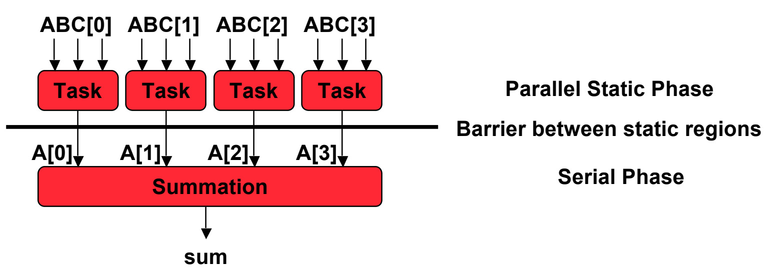


Examining a Task Graph

- Divide graph nodes into levels
  - Put parallel tasks on the same level
  - Put dependent (and therefore serial) tasks on different levels
- Examine each level & classify it . . .
  - Can you use static scheduling on this level?
    - All ~fixed-length tasks
    - ~Fixed number of tasks
  - Phases: should you put a barrier before/after it?
    - Static, with lots of broadcast/reduce-style communication: YES
      - Try to eliminate the need for locks to protect the communication
    - Dynamic, or static with point-to-point communication: NO
    - Static/dynamic scheduling border: PROBABLY
      - Need to be *very* careful if you don't, though

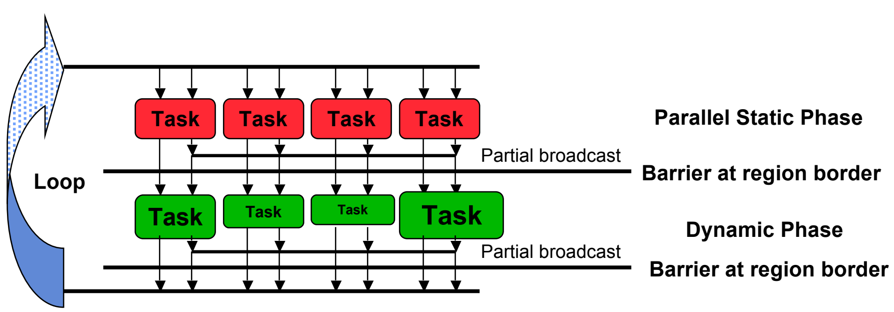
Simple Example

- Back to the simple example again:



More Interesting Example

- This simulation example is more irregular:
  - First phase updates grid points
  - Second phase works with sparse objects in the grid
    - Varying amounts of work per data point

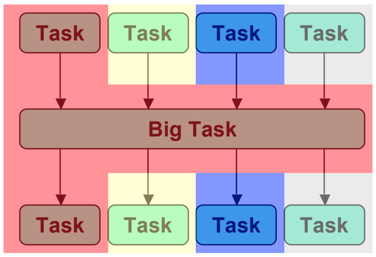


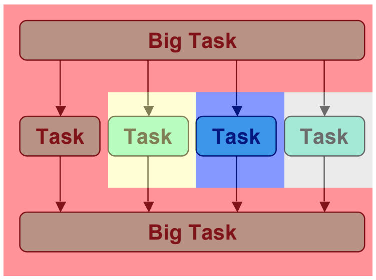
Mapping: Grouping Tasks Together

- OK, so we've got a task dependency graph . . . now what?
- Now need to combine tasks *together*
  - Should have LOTS of tasks in any parallel application
    - ~10x the number of processors
    - If not, we'll probably have load balancing problems
  - Need to *map* to a relatively small number of processors
- The main factor to consider: *communication*
  - Communication is expensive, so let's minimize it!

Mapping Through Time

- Mapping across program phases is simple
  - Try to have processors depend upon *themselves*
    - No communication, locking
  - Fill in unassigned processors as necessary
  - *Examples:*



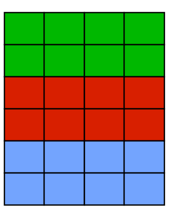


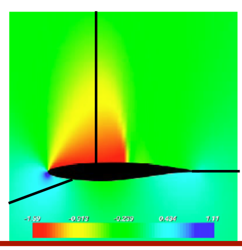
Mapping Across Space

- Dividing up the tasks within a phase is much more work
- Need to think about impact of communication at start/end
  - Which processors are producing inputs?
  - Where do the outputs go?
- May need to think about memory usage *within* phase
  - To improve cache performance
  - To be selective about data copying
- Type of phase affects the mapping
  - *Static:* Try to minimize communication
  - *Dynamic:* Must try to balance work *and* communication
    - Grouping tasks can minimize communication among them
    - But must leave enough groups to allow dynamic load balancing

Blocking

- *Blocking* is applying spatial task mappings to:
  - Tasks created from domain decomposition (data-level parallelism)
  - Data-level parallelism among elements of a *regular array*
- Offers a good example of how to think about mapping
  - Irregular data parallelism can be "blocked" too
  - But all other examples are much harder to illustrate

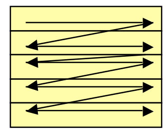




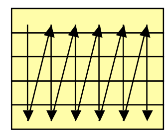
Blocking for Caches I

- Which of the following is faster in C?
  - Your mileage my vary with other languages

```
for (i=0; i < 10000; i++)
  for (j=0; j < 10000; j++)
   sum += a[i][j];
```



```
for (j=0; j < 10000; j++)
  for (i=0; i < 10000; i++)
   sum += a[i][j];
```



Blocking for Caches II

- We want to exploit cache locality in serial programs
  - C stores rows of data together, so . . .
  - Row-major access is better in C
  - Bonus: Also improves memory *page* locality

- We want to exploit locality in parallel programs, too
  - Same advantages as serial code, plus:
  - Processors have separate caches
    - We want them to hold different data, not just copies
    - Want to avoid communication
  - *Moral:* Row-major is *usually* best in parallel C
  - Extra issues
    - Communication
    - Load imbalance

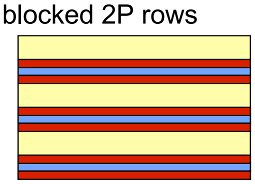
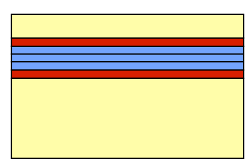
Blocking Rows Together I

- We must often allocate N rows to P processors, N >> P
  - Can allocate interleaved: 1 row/processor
  - Can allocate fully blocked: N/P rows/processor
  - Can allocate partially blocked: 1 < rows/processor < N/P

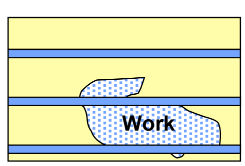
- Must decide best blocking on a case-by-case basis
  - Fully blocked usually best for pure static
    - Keeps neighbors close together
      - Important for lots of algorithms
  - Interleaved/partially blocked best for more dynamic code
    - Distributes entire input array more evenly across processors
    - Provides small chunks for dynamic scheduling
      - More chunks allow better load balancing . . .
      - . . . But don't want to get too small due to scheduling overhead

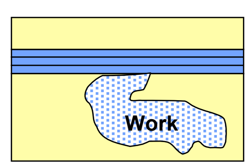
Blocking Rows Together II

- Keeping neighbors together can minimize communication:
  - Nearest neighbor communication: interleaved 2N rows, blocked 2P rows

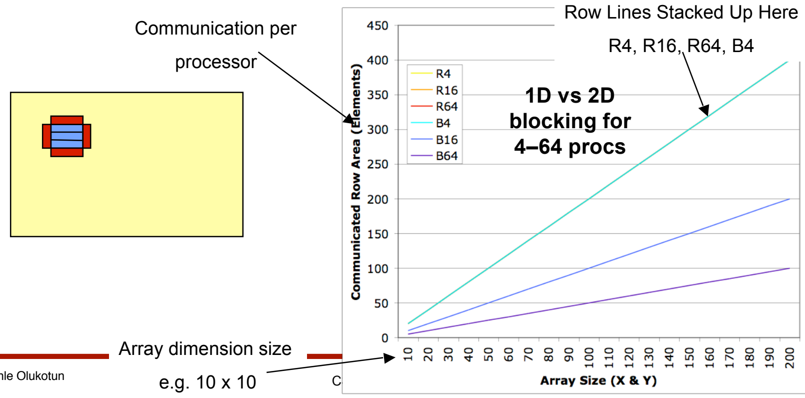
- Breaking rows apart can improve load balance:





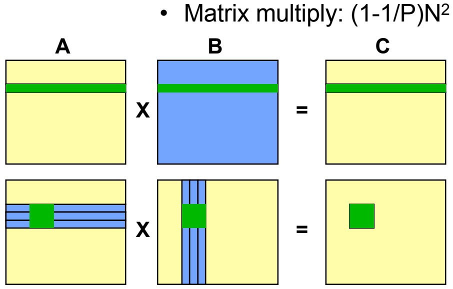
Multi-Dimensional Blocking

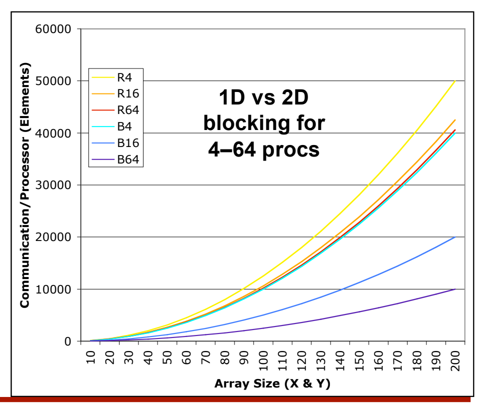
- Many parallel algorithms work best with multi-d "grid" blocking
  - $P^{1/d}$ blocks in each dimension of a d-dimensional grid
  - Reduces edge communication: (2/sqrt(P)) (rows + columns)
    - Although the communication of columns may backfire



Blocking Matrix Multiply

- Can reduce algorithm communication:
  - Output decomposition on Matrix C
  - Matrix multiply: $(1-1/P)N^2$ vs. $(/\text{sqrt}(P) - 1/P)2N^2$

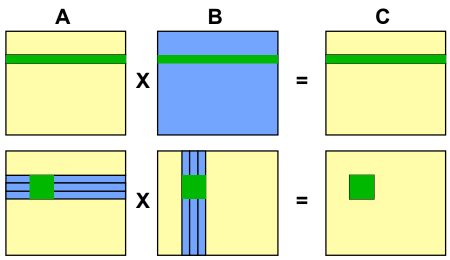




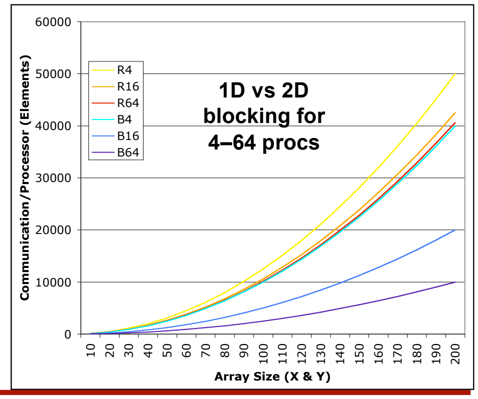
- Each processor (P) gets same partition of matrices A, B and C
- Green = no communication
- Blue = communication

Blocking Matrix Multiply Example

- Can reduce algorithm communication:
  - Output decomposition on Matrix C
  - Matrix multiply: $(1-1/P)N^2$ vs. $(1/\text{sqrt}(P) - 1/P)2N^2$



– 16 processors
  - 1D: $(1- 1/16)\ N^2 = 15/16N^2$
  - 2D: $(1/4 - 1/16)\ 2\ N^2 = 6/16N^2$



Implementing Multi-Dimensional Blocking

- So how do we do it?
  - Need to "make your own" in both pthreads and OpenMP

- Code nominally uses a 4-level loop:

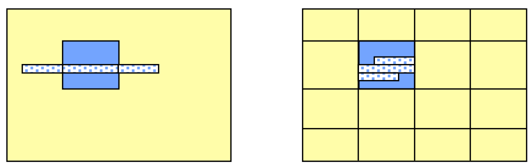
```
PARALLEL for (p_x=0; p_x < sqrt(NUM_PROCS); p_x++)
PARALLEL for (p_y=0; p_y < sqrt(NUM_PROCS); p_y++){
     /* Calculate a block on each processor */
     SERIAL for (x=0; x < BLOCK_WIDTH; x++)
     SERIAL for (y=0; y < BLOCK_HEIGHT; y++)
             CalculateResult( A[p_x][p_y][x][y] );}
```

- May need to flatten the two parallel loops to a single loop:

  Parallel XY loop, MxN blocks
  X = XY / M;
  Y = XY % M;
  Serial loops as before . . .

Blocking and Spatial Locality

- Breaking rows can decrease cache locality
  - Can end up with cache lines straddling column breaks
  - Also hurts with page locality, increasing TLB misses
- Make sure block size aligned with cache line
- So use 2xd-dimensional arrays
  - Every dimension is broken into two subdimensions:
    - Parallel processor ID number
    - Serial dimension
  - Create a d-dimensional array for each block
  - Effectively independent arrays for *each* processor

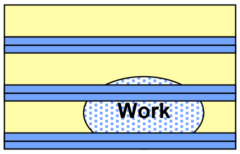


Blocking and Spatial Locality II

- So use 2xd-dimensional arrays
  - Every dimension is broken into two subdimensions:
    - Parallel processor ID number
    - Serial dimension
  - Create a d-dimensional array for each block
- Original code:
  - A[x][y]
- Transformed code:
  - **`A[processor dimensions][linear dimensions]`**
  - **`A[x-proc][y-proc][x-linear][y-linear]`** **`or`**
    `. . .`
    `A[x/BLOCK_SIZE][y/BLOCK_SIZE][x%BLOCK_SIZE][y %BLOCK_SIZE]`
  - with each processor working on one particular block
  - Putting "processor number" dimensions first groups each processor's accesses to a contiguous area of memory

Blocking and Load Imbalance

- Blocking can lead to load imbalance
  - Amount of work per matrix element varies e.g. (LU factorization)
- Block-cyclic distributions:
  - Tasks with *predictably uneven* timing can be spread across processors *without* dynamic allocation
  - Interleave processors in “light” and “heavy” areas of data to balance out on average
  - Static “load rebalancing”



Reductions

- One of the banes of parallelism is a *reduction* in dimensionality
  - Go from N dimensions to N-1, N-2, . . . 0
  - Dot products are the most common example
    - a[i] = a[i] + b[j] x c[j]
- *Divide on output* for target dimension > 0
- Single output, *associative* reduction
  - Combine to P elements
    - Do as much of the reduction in parallel as possible
  - Do final step in serial (small P) or in a parallel tree (large P)
- Single output, *non-associative* reduction
  - It’s serial, so try to overlap *parts* of tasks
  - Good place to apply pipeline parallelism!

Reductions in OpenMP

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
- The reduction clause has this syntax:
  reduction (<op> :<variable>)
- Operators
  - \+ Sum
  - * Product
  - &, |, ^ Bitwise and, or , exclusive or
  - &&, || Logical and, or

OMP Code with Reduction Clause

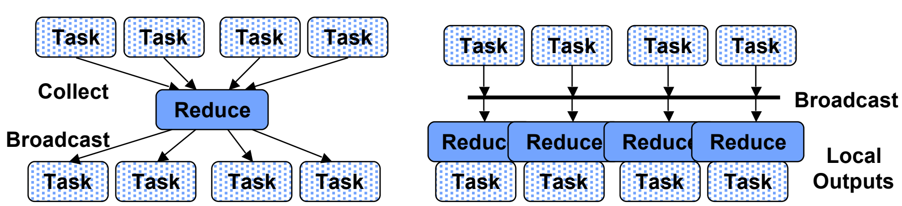
```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
       private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Reduction Alternatives in OpenMP

- OpenMP non-associative reduction gives some choices
  - Use OpenMP `ordered` block
    - Forces all processors to do reduction step in-order
  - Use OpenMP `barrier` + `single` block
    - Lets only one processor do the reduction serially
  - Or just end the parallel region and do it serially

Replication of Work

- Sometimes it's better to add tasks than to communicate
  - A small "global" value generation
    - Reduction results then used by many processors
  - Need to carefully consider if cost of extra work is worth it
    - Replicated work doesn't help you speed up!
    - Must be worth the communication savings!



Summary & A Look Ahead

- We went through the process of parallelizing an application:
  - Identify possible parallel tasks
  - Break down tasks and find dependence graph
  - Find phases in dependence graph and classify them
  - Map tasks to processors
    - In time (simple)
    - In space (more complex, may need to block data)
- Will next look at measuring and evaluating performance!
  - Some common parallel programming issues
  - The many definitions of “speedup” in parallel systems
  - Analyzing scalability of applications