High-Performance Scientific Computing
Lecture 13: Parallel Patterns

MATH-GA 2011 / CSCI-GA 2945 · December 5, 2012

Today

Tool of the day: 3D Visualization

Parallel Patterns

Bits and pieces

- HW6: soon
- Dec 12: No class–good luck on finals!
- Dec 17?/18?/**19**: Project presentations
  - Will announce precise date, watch email
- Project guidelines posted
- Need help with project? Ask/come see us!
- Class evaluations

Outline

Tool of the day: 3D Visualization

Parallel Patterns

3D vis demo time

Visualization demo

Software links:

- libsilo (LLNL “WCI”, BSD license)
- VisIt (LLNL “WCI”, BSD license)

Alternative:

- Paraview (KitWare/LANL, BSD license)
- TecPlot ($$$)

Outline

Tool of the day: 3D Visualization

Parallel Patterns
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Outline

Tool of the day: 3D Visualization

Parallel Patterns

- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Partition

$$y_i = f_i(x_{i-1}, x_i, x_{i+1})$$

where $i \in \{1, \ldots, N\}$.

Partition

$$y_i = f_i(x_{i-1}, x_i, x_{i+1})$$

where $i \in \{1, \ldots, N\}$.

Includes straightforward generalizations to dependencies on a larger (but not $O(P)$-sized!) set of neighbor inputs.

Partition

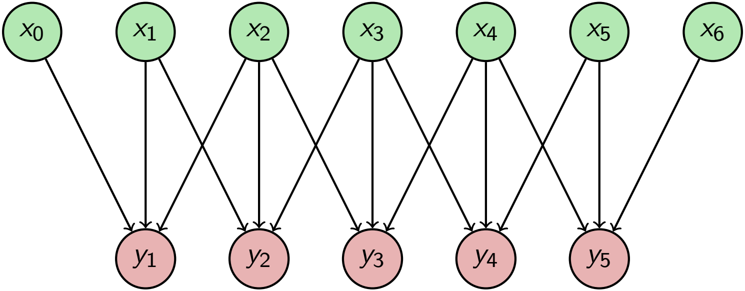
$$y_i = f_i(x_{i-1}, x_i, x_{i+1})$$

where $i \in \{1, \ldots, N\}$.

Includes straightforward generalizations to dependencies on a larger (but not $O(P)$-sized!) set of neighbor inputs.

Point: Processor i *owns* x_i. ("owns" = is "responsible for updating")

Partition: Graph



Mapping to Mechanisms

- OpenMP?

Mapping to Mechanisms

- OpenMP?
- MPI?

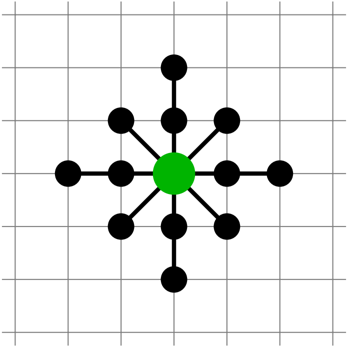
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

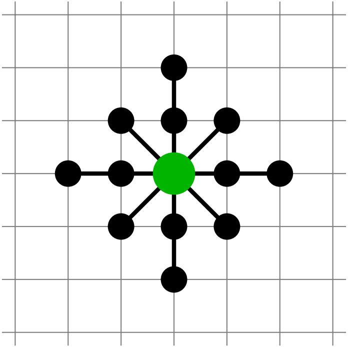
Mapping to Mechanisms: Stencils



Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

Mapping to Mechanisms: Stencils

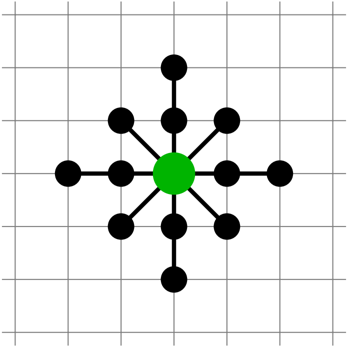


Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential

Mapping to Mechanisms: Stencils

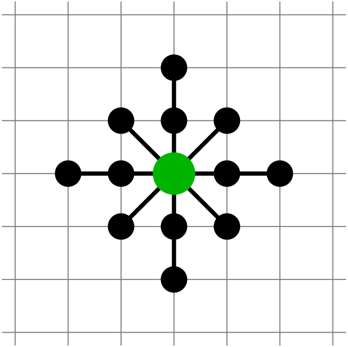


Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential
- OpenMP?

Mapping to Mechanisms: Stencils

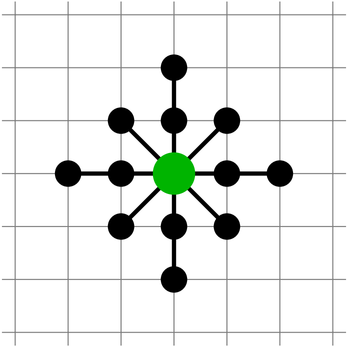


Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential
- OpenMP?
- MPI?

Mapping to Mechanisms: Stencils

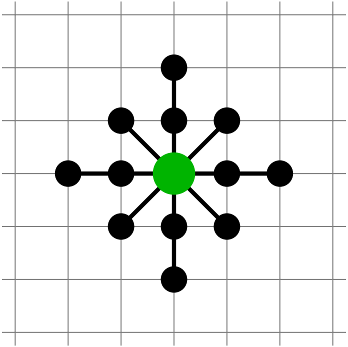


Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?

Mapping to Mechanisms: Stencils

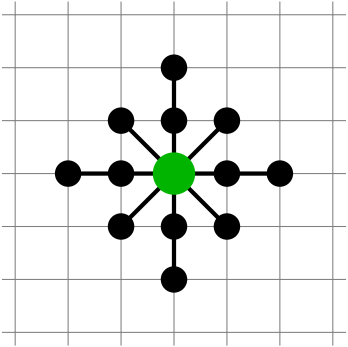


Common example ("5-point stencil"):

$$u^{n+1}_{i,j} = \frac{1}{h^2}(-4u^n_{i,j} + u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1})$$

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?

Mapping to Mechanisms: Stencils



Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?

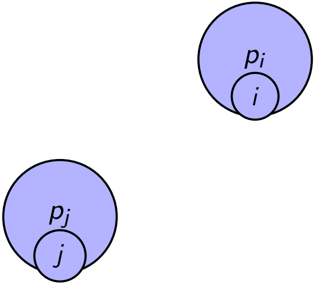
What if there's geometry?

Partition: Issues

- Same computation often repeated many times
  - As time steps in a simulation
  - Until ‘convergence’
- $\rightarrow$ Synchronization?
- Main structures: Array (image, grid), Graph (mesh)
- Performance impact of partition?
- Granularity?
- Only useful when the computation is mainly local
- Load Balancing: Thorny issue (next)

Rendezvous Trick

- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

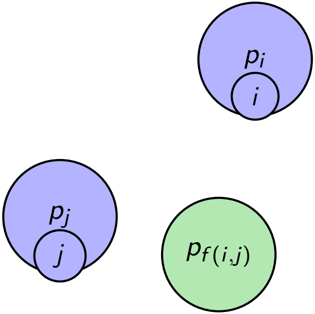


Rendezvous Trick

- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

For f: think 'hash function'.

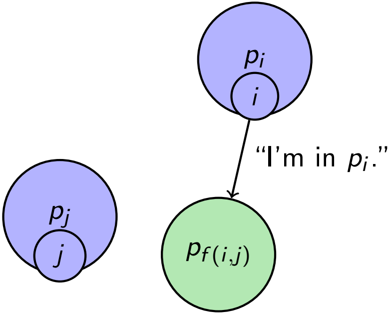


Rendezvous Trick

- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

For f: think 'hash function'.

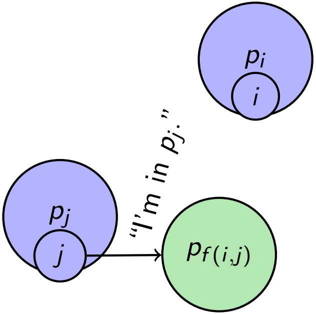


Rendezvous Trick

- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

For f: think 'hash function'.

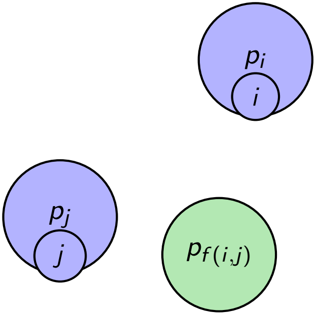


Rendezvous Trick

- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

For f: think 'hash function'.

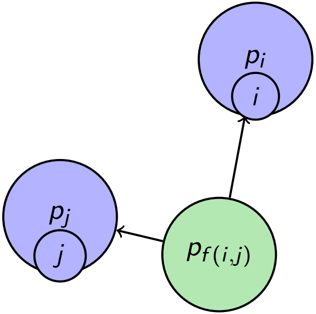


Rendezvous Trick

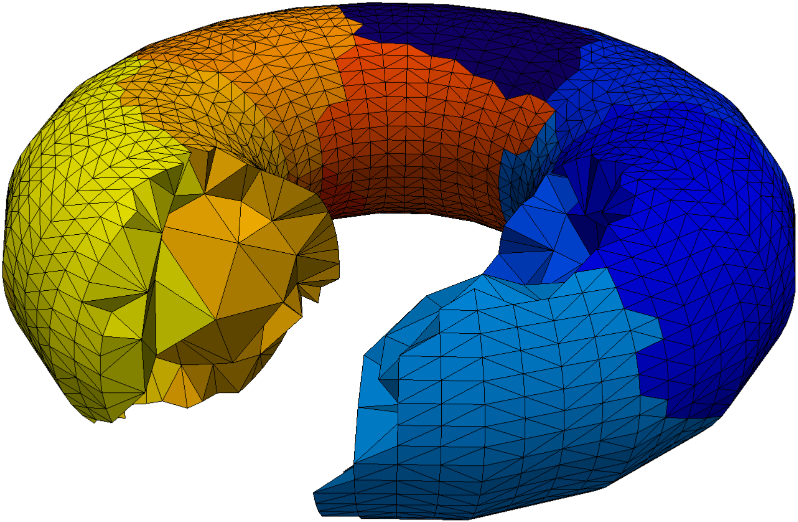
- Assume an irregular partition.
- Assume problem components i, j on unknown partitions p_i, p_j need to communicate.
- How can p_i find p_j (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

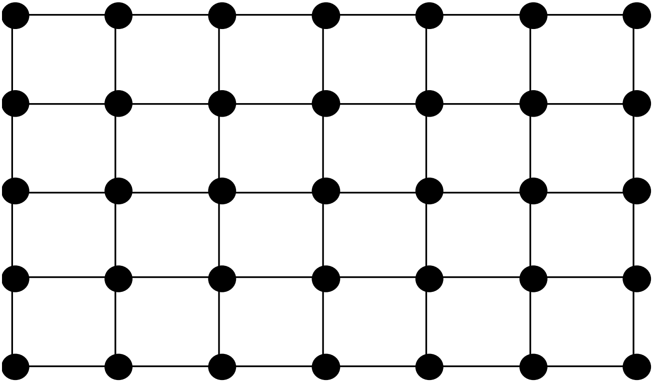
For f: think 'hash function'.



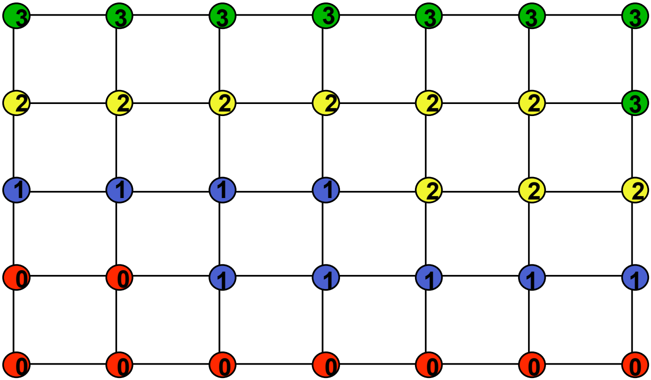
Partitioning for neighbor communication



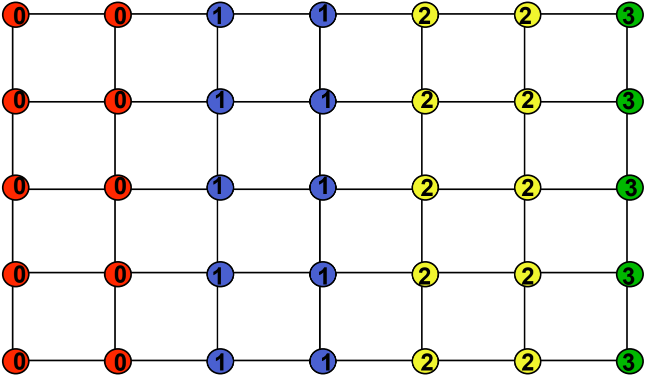
Example



Example

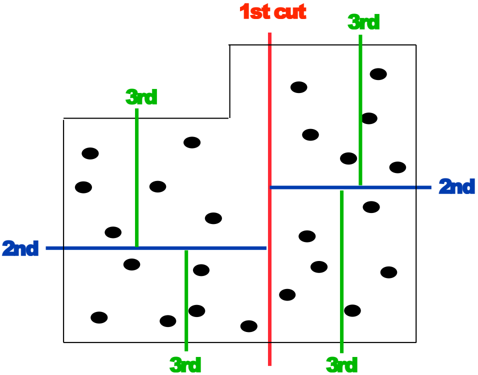


Example



A simple strategy

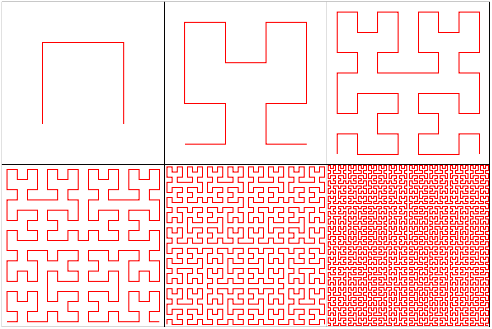
Recursive Coordinate Bisection ('RCB') [Berger, Bokhari '87]



- ⊕ Simple
- ⊕ Easy to update for changed geometry ('incremental')
- ⊖ Easy to fool

E. Boman, K. Devine (Sandia)

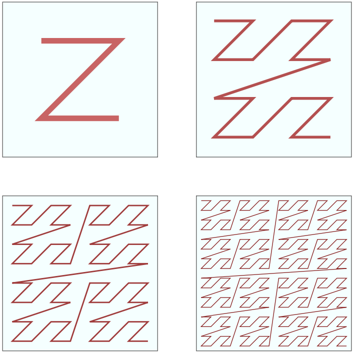
Space-filling curves



Hilbert curve

Wikipedia

Space-filling curves

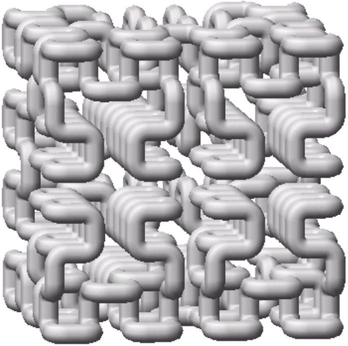


Morton curve ("Z curve")

Easily obtained by bit interleaving!

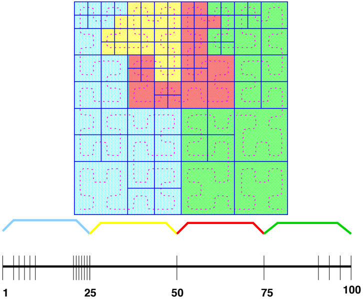
Wikipedia

Space-filling curves



Carlo H. Sequin, UC Berkeley / Wikipedia

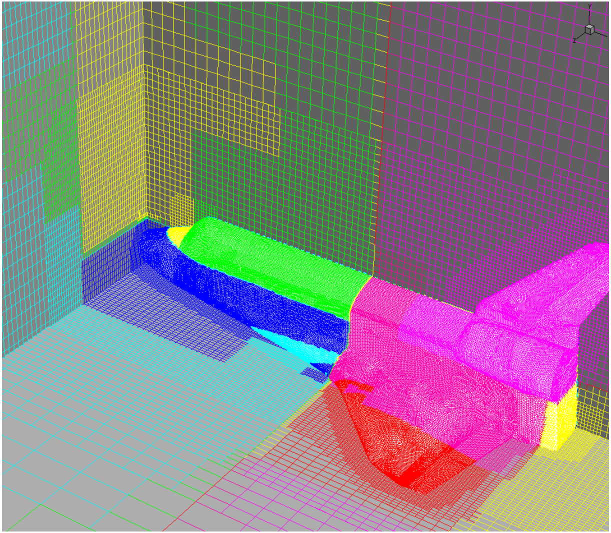
Space-filling curves



- Simple, even for adaptive meshes
- Weight-able
- Cache-happy
- Easy to update for changed geometry ('incremental')
- Communication volume?

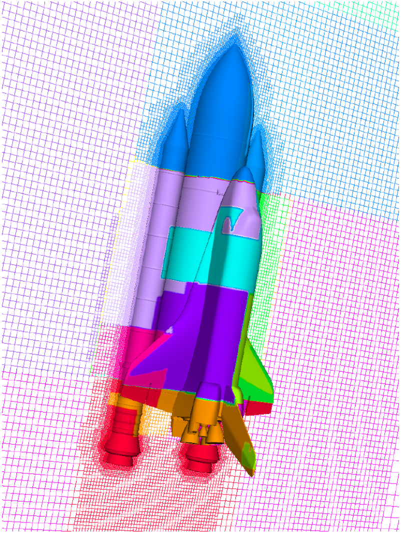
M. Berger

Space-filling curves: Examples



M. Berger, M. Aftosmis

Space-filling curves: Examples



M. Berger, M. Aftosmis

Partitioning: Objectives

Main goals:

- Even distribution of work
- Minimize neighbor communication

Criteria:

- Cheap! (General problem: NP-complete)
- Incremental
- Partitioning itself is parallel

Partitioning: Objectives

Main goals:

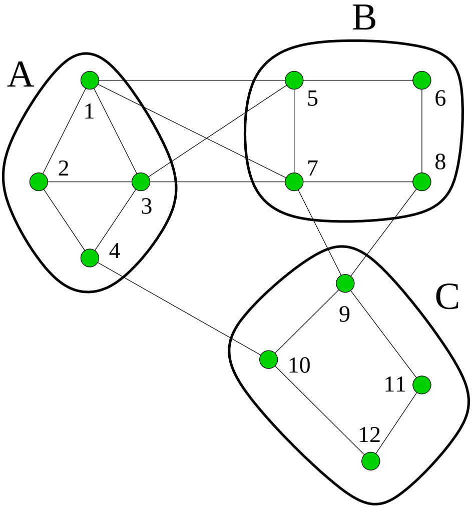
- Even distribution of work
- Minimize neighbor communication

Criteria:

- Cheap! (General problem: NP-complete)
- Incremental
- Partitioning itself is parallel

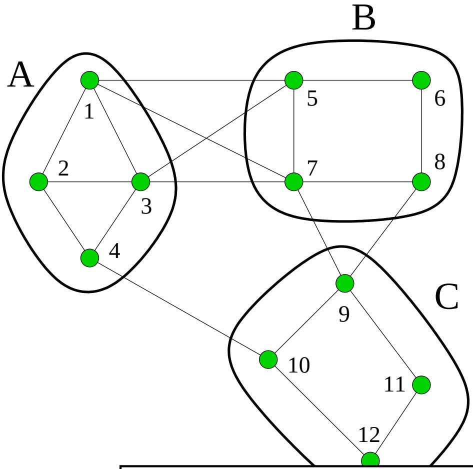
What if we *don't* have geometry/coordinates?

Chopping up the communication graph



K. Schloegel, G. Karypis, V. Kumar ‘00

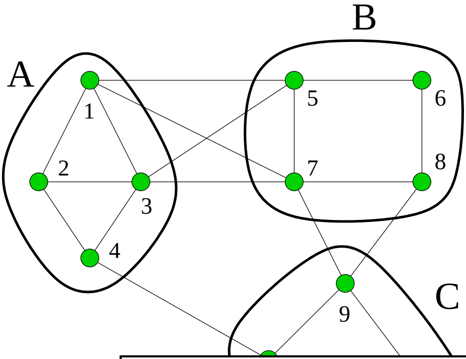
Chopping up the communication graph



Great model? How often do we send vertex 1 to B?

K. Schloegel, G. Karypis, V. K

Chopping up the communication graph



Great model? How often do we send vertex 1 to B?

Perhaps: assign weight to vertices, edges

K. Schloegel, G. Karypis, V. K

Spectral partitioning demo

Partitioning

Metis demo

Finer points

- What if # inputs $\neq$ # outputs?
- Might want to balance multiple objectives
  - Types of work
  - Types of communication

Software packages to look for:

- Zoltan (free, LGPL)
- PT-Scotch (free, copyleft)
- Metis (free to use, proprietary, some source available)

Finer points

- What if # inputs $\neq$ # outputs? ($\rightarrow$ hypergraphs)
- Might want to balance multiple objectives
  - Types of work
  - Types of communication

Software packages to look for:

- Zoltan (free, LGPL)
- PT-Scotch (free, copyleft)
- Metis (free to use, proprietary, some source available)

Outline

Tool of the day: 3D Visualization

Parallel Patterns

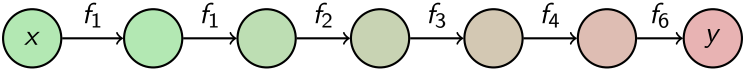
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Pipelined Computation

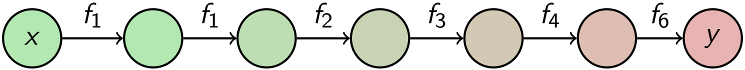
$$
\begin{aligned}
y &= f_N(\cdots f_2(f_1(x))\cdots) \\
&= (f_N \circ \cdots \circ f_1)(x)
\end{aligned}
$$

where N is fixed.

Pipelined Computation: Graph



Pipelined Computation: Graph



Processor Assignment?

Pipelined Computation: Examples

- Image processing
- Any multi-stage algorithm
  - Pre/post-processing or I/O
- Out-of-Core algorithms

Specific simple examples:

- Sorting (insertion sort)
- Triangular linear system solve ('backsubstitution')
  - Key: Pass on values as soon as they're available

(will see more efficient algorithms for both later)

Pipelined Computation: Issues

- Non-optimal while pipeline fills or empties
- Often communication-inefficient
  - for large data
- Needs some attention to synchronization, deadlock avoidance
- Can accommodate some asynchrony
  But don't want:
  - Pile-up
  - Starvation

Mapping to Mechanisms

- OpenMP?

Mapping to Mechanisms

- OpenMP?
- MPI?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

Outline

Tool of the day: 3D Visualization

Parallel Patterns
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Reduction

$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

where N is the input size.

Reduction

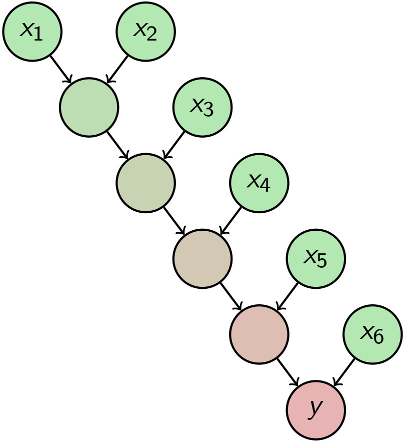
$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

where N is the input size.

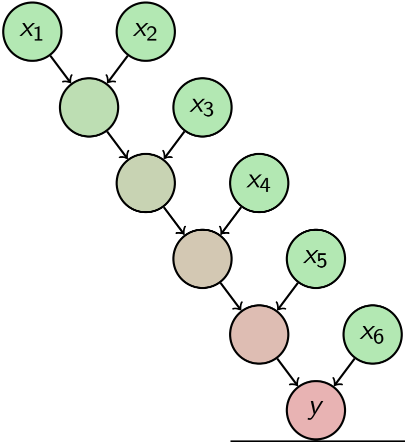
Also known as...

- Lisp/Python function `reduce` (Scheme: `fold`)
- C++ STL `std::accumulate`

Reduction: Graph

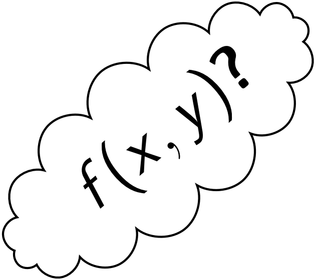


Reduction: Graph



Painful! Not parallelizable.

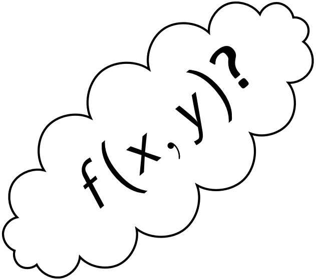
Approach to Reduction



Can we do better?

"Tree" very imbalanced. What property of f would allow 'rebalancing'?

Approach to Reduction



Can we do better?

"Tree" very imbalanced. What property of f would allow 'rebalancing'?

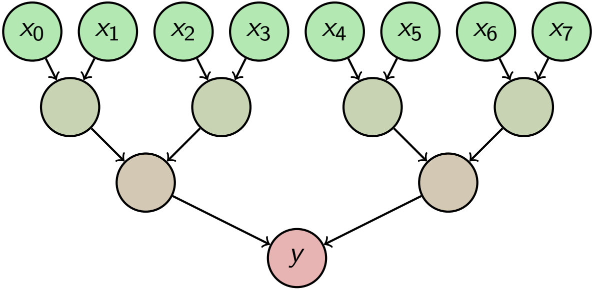
$$f(f(x,y),z) = f(x, f(y,z))$$

Looks less improbable if we let $x \circ y = f(x,y)$:

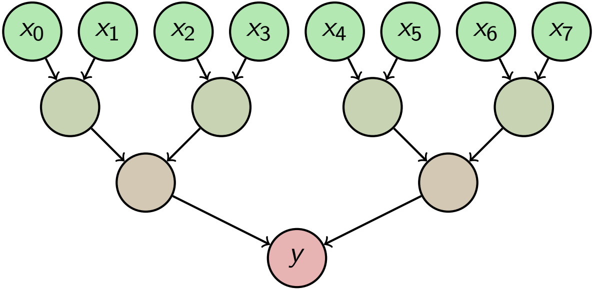
$$x \circ (y \circ z)) = (x \circ y) \circ z$$

Has a very familiar name: *Associativity*

Reduction: A Better Graph



Reduction: A Better Graph



Processor allocation?

Mapping to Mechanisms

- Single threads?

Mapping to Mechanisms

- Single threads?
- OpenMP?

Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?

Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

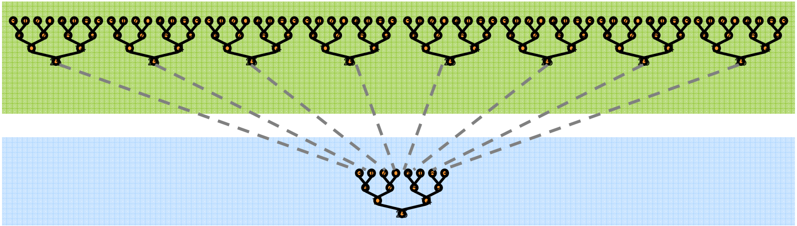
Mapping Reduction to the GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.

With material by M. Harris (Nvidia Corp.)

Mapping Reduction to the GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.



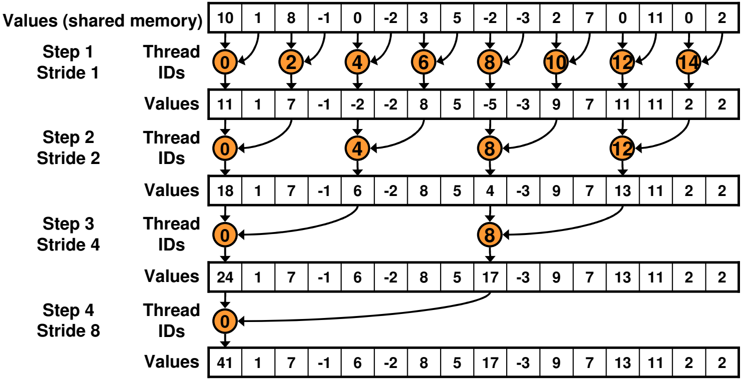
In particular: Use multiple grid invocations to achieve inter-workgroup synchronization.

With material by M. Harris (Nvidia Corp.)

Kernel V1

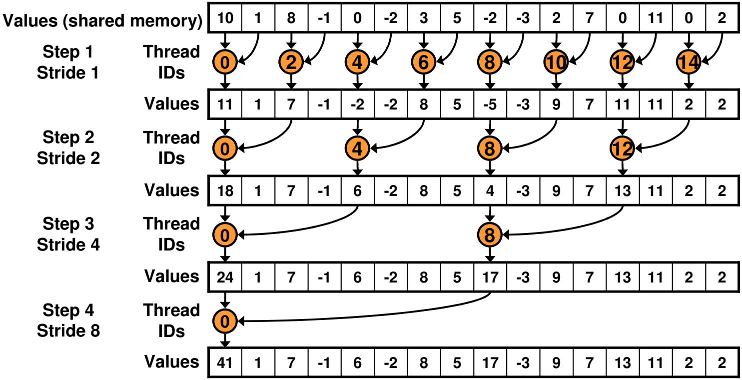
```
__kernel void reduce0( __global T *g_idata, __global T *g_odata,
    unsigned int n, __local T* ldata)
{
    unsigned int lid = get_local_id (0);
    unsigned int i = get_global_id (0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier (CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=1; s < get_local_size (0); s *= 2)
    {
        if ((lid % (2*s)) == 0)
            ldata[lid] += ldata[lid + s];
        barrier (CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
}
```

Interleaved Addressing



With material by M. Harris (Nvidia Corp.)

Interleaved Addressing



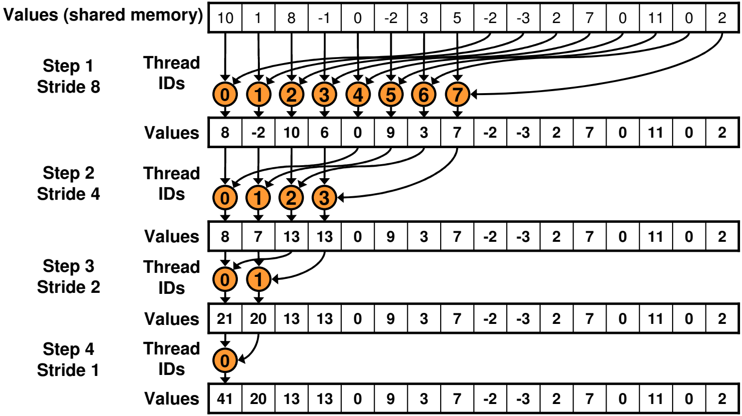
Issue: Slow modulo, Divergence

With material by M. Harris (Nvidia Corp.)

Kernel V2

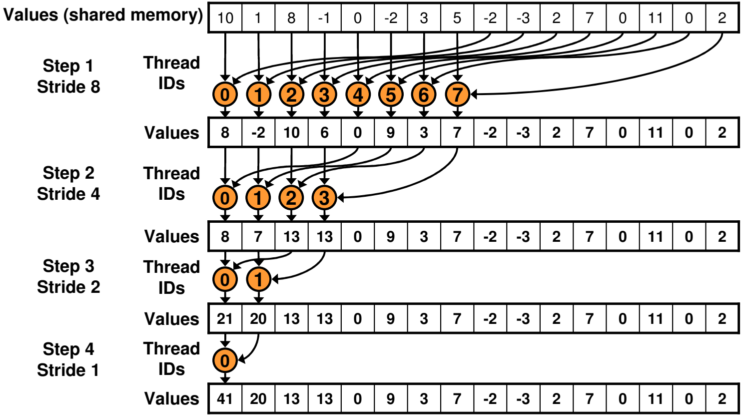
```
__kernel void reduce2( __global T *g_idata, __global T *g_odata,
    unsigned int n, __local T* ldata)
{
    unsigned int lid = get_local_id(0);
    unsigned int i = get_global_id(0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=get_local_size(0)/2; s>0; s>>=1)
    {
        if (lid < s)
            ldata[lid] += ldata[lid + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[get_local_size(0)] = ldata[0];
}
```

Sequential Addressing



With material by M. Harris (Nvidia Corp.)

Sequential Addressing



Better! But still not "efficient".

Only half of all work items after first round, then a quarter, . . .

With material by M. Harris (Nvidia Corp.)

Recap: Parallel Complexity

Distinguish:

- Time on T processors: T_P
- **Step Complexity/Span** T_∞: Minimum number of steps taken if an infinite number of processors are available
- Work per step S_t
- **Work Complexity/Work** $T_1 = \sum_{t=1}^{T_\infty} S_t$: Total number of operations performed
- **Parallelism** T_1/T_∞: average amount of work along span
  - $P > T_1/T_\infty$ doesn't make sense.

Algorithm-specific!

Parallel Complexity for Reduction

Number of Items N
Actual work to be done: $W = O(N)$ additions.

Step Complexity: Let $d = \lceil \log_2 N \rceil$. Then $T_\infty = d$, $S_t = O(2^{d-t})$.

Work Complexity:

$$T_1 = \sum_{t=1}^{T} S_t = O\left(\sum_{t=1}^{T} 2^{d-t}\right) = O(2^d) = O(N)$$

Parallel Complexity for Reduction

Number of Items N
Actual work to be done: $W = O(N)$ additions.

Step Complexity: Let $d = \lceil \log_2 N \rceil$. Then $T_\infty = d$, $S_t = O(2^{d-t})$.

Work Complexity:

$$T_1 = \sum_{t=1}^{T} S_t = O\left(\sum_{t=1}^{T} 2^{d-t}\right) = O(2^d) = O(N)$$

"Work-efficient:" $T_1 \sim W$.

Greedy Scheduling

Theorem (Graham '68, Brent '75)

A parallel algorithm with span T_∞ and work complexity T_1 can be executed on a shared-memory machine with P processors in no more than

$$T_P \leq \frac{T_1}{P} + T_\infty$$

steps.

Observations:

- Think of T_∞ as the length of the "critical path".
- The first summand can be made to go away by increasing P.
- Only valid for shared-memory.

Greedy Scheduling

Theorem (Graham '68, Brent '75)

A parallel algorithm with span T_∞ and work complexity T_1 can be executed on a shared-memory machine with P processors in no more than

$$T_P \leq \frac{T_1}{P} + T_\infty$$

steps.

Observations:

- Think of T_∞ as the leng
- The first summand can
- Only valid for shared-me

Estimate for $P = 1$?

Proof sketch?

What about reduction?

What is P for a GPU?

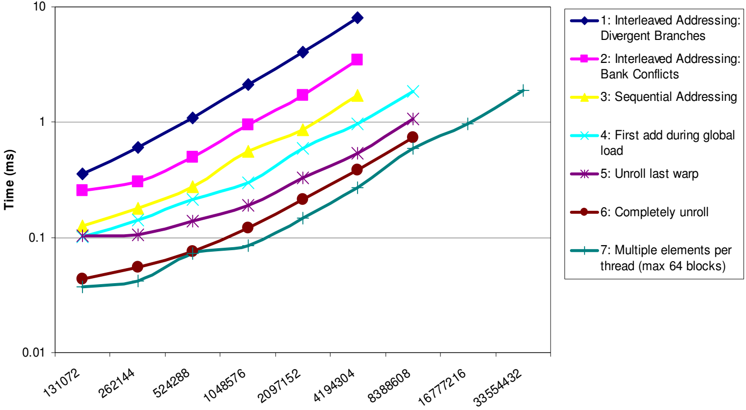
Kernel V3 Part 1

```
__kernel void reduce6( __global T *g_idata, __global T *g_odata,
    unsigned int n, volatile __local T* ldata)
{
    unsigned int lid = get_local_id (0);
    unsigned int i = get_group_id(0)*(
          get_local_size (0)*2) + get_local_id (0);
    unsigned int gridSize = GROUP_SIZE*2*get_num_groups(0);
    ldata[lid] = 0;

    while (i < n)
    {
        ldata[lid] += g_idata[i];
        if (i + GROUP_SIZE < n)
            ldata[lid] += g_idata[i+GROUP_SIZE];
        i += gridSize;
    }
    barrier (CLK_LOCAL_MEM_FENCE);
```

Kernel V3 Part 2

```
    if (GROUP_SIZE >= 512)
    {
      if (lid < 256) { ldata[lid] += ldata[lid + 256]; }
      barrier(CLK_LOCAL_MEM_FENCE);
    }
    // ...
    if (GROUP_SIZE >= 128)
    { /* ... */ }

    if (lid < 32)
    {
        if (GROUP_SIZE >= 64) { ldata[lid] += ldata[lid + 32]; }
        if (GROUP_SIZE >= 32) { ldata[lid] += ldata[lid + 16]; }
        // ...
        if (GROUP_SIZE >= 2) { ldata[lid] += ldata[lid + 1]; }
    }

    if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
}
```

Performance Comparison



With material by M. Harris (Nvidia Corp.)

Reduction: Examples

- Sum, Inner Product, Norm
  - Occurs in iterative methods
- Minimum, Maximum
- Data Analysis
  - Evaluation of Monte Carlo Simulations
- List Concatenation, Set Union
- Matrix-Vector product (but...)

Reduction: Issues

- When adding: floating point cancellation?
- Serial order goes faster: can use registers for intermediate results
- Requires availability of neutral element
- GPU-Reduce: Optimization sensitive to data type

Outline

Tool of the day: 3D Visualization

Parallel Patterns
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Map-Reduce

Sounds like this:

$$y = f(\cdots f(f(g(x_1), g(x_2)), g(x_3)), \ldots, g(x_N))$$

where N is the input size.

- Lisp naming, again
- Mild generalization of reduction

Map-Reduce

But no. Not even close.

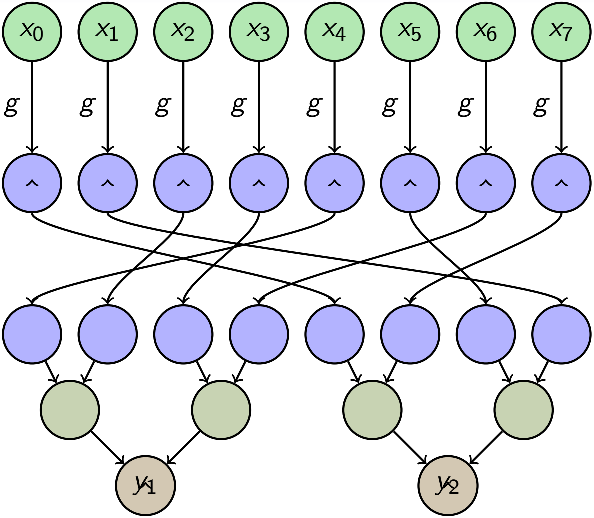
Sounds like this:

$$y = f(\cdots f(f(g(x_1), g(x_2)), g(x_3)), \ldots, g(x_N))$$

where N is the input size.

- Lisp naming, again
- Mild generalization of reduction

Map-Reduce: Graph



MapReduce: Discussion

MapReduce $\geq \texttt{map} + \texttt{reduce}$:

- Used by Google (and many others) for large-scale data processing
- Map generates $(\texttt{key}, \texttt{value})$ pairs
  - Reduce operates only on pairs with *identical keys*
  - Remaining output sorted by key
- Represent all data as character strings
  - User must convert to/from internal repr.
- Messy implementation
  - Parallelization, fault tolerance, monitoring, data management, load balance, re-run “stragglers”, data locality
- Works for Internet-size data
- Simple to use even for inexperienced users

MapReduce: Examples

- String search
- (e.g. URL) Hit count from Log
- Reverse web-link graph
  - desired: (`target URL`, `sources`)
- Sort
- Indexing
  - desired: (`word`, `document IDs`)
- Machine Learning, Clustering, . . .

Outline

Tool of the day: 3D Visualization

Parallel Patterns

- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

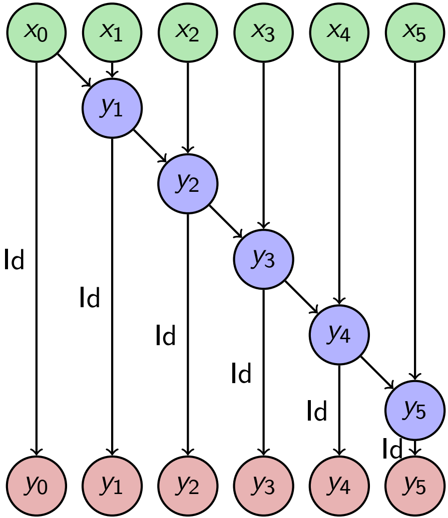
Scan

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= f(y_1, x_2) \\ \vdots &= \vdots \\ y_N &= f(y_{N-1}, x_N) \end{aligned}$$

where N is the input size. (Think: N large, $f(x, y) = x + y$)

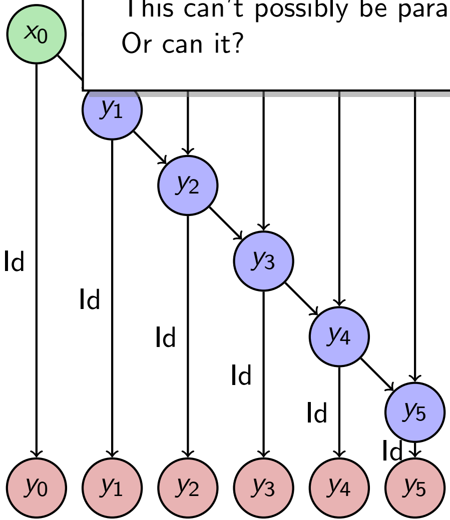
- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

Scan: Graph



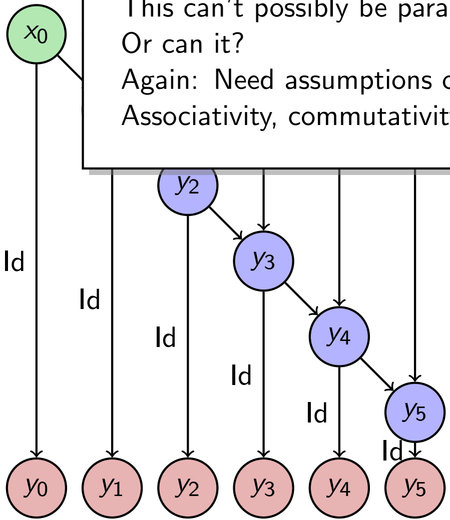
Scan: Graph

This can't possibly be parallelized.
Or can it?

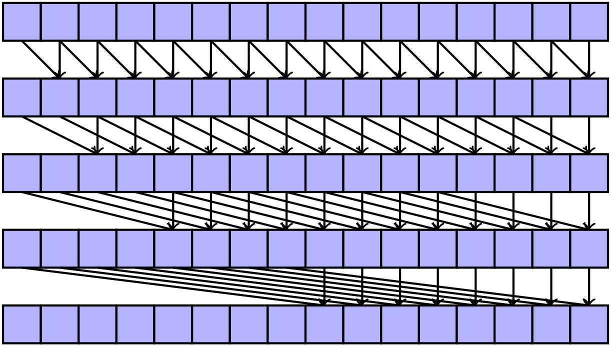


Scan: Graph

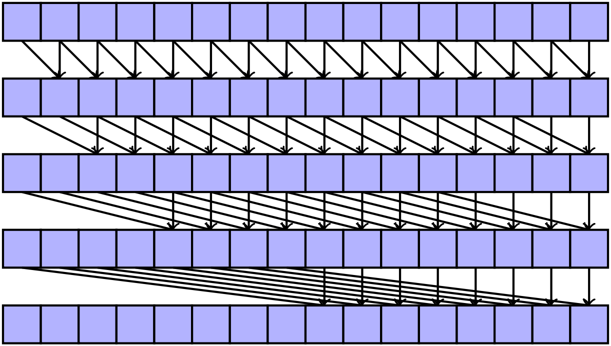
This can't possibly be parallelized.
Or can it?
Again: Need assumptions on f.
Associativity, commutativity.



Scan: Implementation



Scan: Implementation



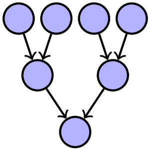
Work-efficient?

Scan: Implementation II

Two sweeps: Upward, downward,
both tree-shape

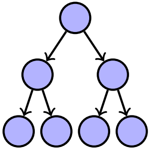
On upward sweep:

- Get values `L` and `R` from left and right child
- Save L in local variable `Mine`
- Compute `Tmp` = `L` + `R` and pass to parent



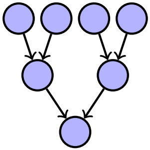
On downward sweep:

- Get value `Tmp` from parent
- Send `Tmp` to left child
- Sent `Tmp+Mine` to right child



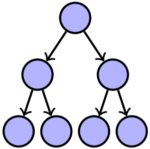
Scan: Implementation II

Two sweeps: Upward, downward,
both tree-shape



On upward sweep:

- Get values L and R from left and right child
- Save L in local variable Mine
- Compute $\texttt{Tmp} = \texttt{L} + \texttt{R}$ and pass to parent



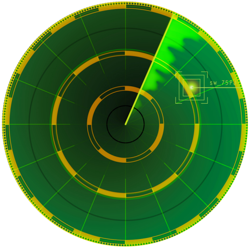
On downward sweep:

- Get value Tmp from parent
- Send Tmp to left chi
- Sent Tmp+Mine to r

Work-efficient?
Span rel. to first attempt?

Scan: Examples

- Anything with a loop-carried dependence
- One row of Gauss-Seidel
- One row of triangular solve
- Segment numbering if boundaries are known
- Low-level building block for many higher-level algorithms algorithms
- FIR/IIR Filtering
- G.E. Blelloch: [Prefix Sums and their Applications]



Scan: Issues

- Subtlety: Inclusive/Exclusive Scan
- Pattern sometimes hard to recognize
  - But shows up surprisingly often
  - Need to prove associativity/commutativity
- Useful in Implementation: algorithm cascading
  - Do sequential scan on parts, then parallelize at coarser granularities

Mapping to Mechanisms

- OpenMP?

Mapping to Mechanisms

- OpenMP?
- MPI?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

Challenge

Sort (fixed-size) integers using scan

Outline

Tool of the day: 3D Visualization

Parallel Patterns

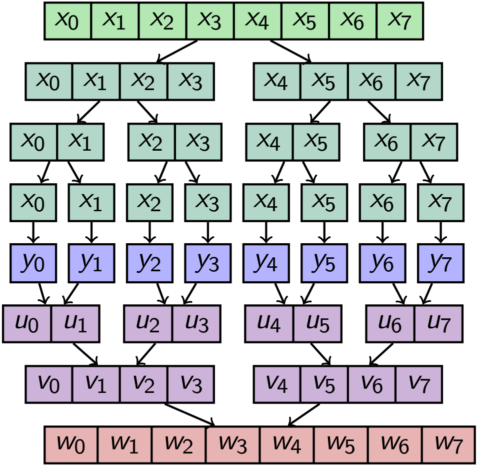
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

Divide and Conquer

$$y_i = f_i(x_1, \ldots, x_N)$$

for $i \in \{1, \ldots, M\}$.

Main purpose: A way of partitioning up fully dependent tasks.



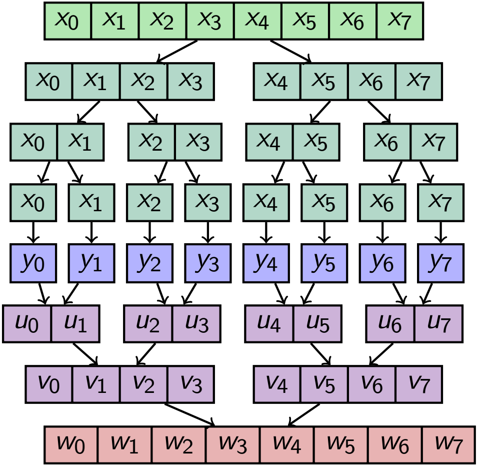
Divide and Conquer

$$y_i = f_i(x_1, \ldots, x_N)$$

for $i \in \{1, \ldots, M\}$.

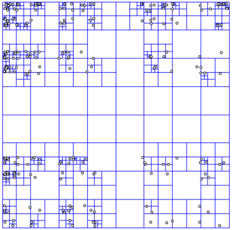
Main purpose: A way of partitioning up fully dependent tasks.

Processor allocation?



Divide and Conquer: Examples

- GEMM, TRMM, TRSM, GETRF (LU)
- FFT
- Sorting: Bucket sort, Merge sort
- *N*-Body problems (Barnes-Hut, FMM)
- Adaptive Integration



More fun with work and span:
D&C analysis lecture

Mapping to Mechanisms

- OpenMP?

Mapping to Mechanisms

- OpenMP?
- MPI?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

Divide and Conquer: Issues

- "No idea how to parallelize that"
  - $\rightarrow$ Try D&C
- Non-optimal during partition, merge
  - But: Does not matter if deep levels do heavy enough processing
- Subtle to map to fixed-width machines (e.g. GPUs)
  - Varying data size along tree $\rightarrow$ Scan!
- Bookkeeping nontrivial for non-2^n sizes
- Side benefit: D&C is generally cache-friendly

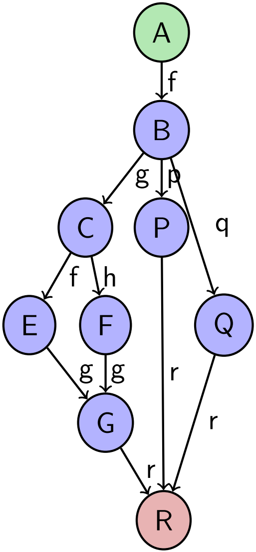
Outline

Tool of the day: 3D Visualization

Parallel Patterns
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies

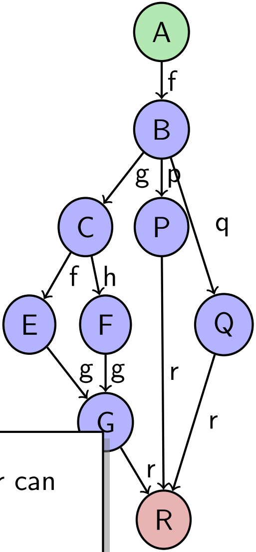
General Dependency Graphs

$B = f(A)$
$C = g(B)$
$E = f(C)$
$F = h(C)$
$G = g(E,F)$
$P = p(B)$
$Q = q(B)$
$R = r(G,P,Q)$



General Dependency Graphs

B = f(A)
C = g(B)
E = f(C)
F = h(C)
G = g(E,F)
P = p(B)
Q = q(B)
R = r(G,P,Q)



Great: All patterns discussed so far can be reduced to this one.

Mapping to Mechanisms

- OpenMP?

Mapping to Mechanisms

- OpenMP?
- MPI?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?

Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?

Cilk

```
cilk int fib (int n)
{
  if (n < 2) return n;
  else
  {
    int x, y;

    x = spawn fib (n-1);
    y = spawn fib (n-2);

    sync;

    return (x+y);
  }
}
```

Features:

- Adds keywords `spawn`, `sync`, (`inlet`, `abort`)
- Remove keywords → valid (seq.) C

Timeline:

- Developed at MIT, starting in ‘94
- Commercialized in ‘06
- Bought by Intel in ‘09
- Available in the Intel Compilers

Cilk

```
cilk int fib (int n)
{
  if (n < 2) return n;
  else
  {
    int x, y;

    x = spawn fib (n-1);
    y = spawn fib (n-2);

    sync;

    return (x+y);
  }
```

Efficient implementation?

Features:

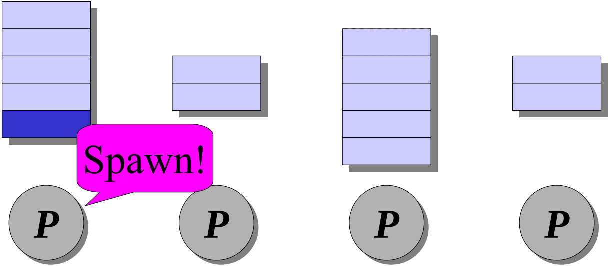
- Adds keywords `spawn`, `sync`, (`inlet`, `abort`)
- Remove keywords → valid (seq.) C

Timeline:

- Developed at MIT, starting in '94
- Commercialized in '06
- Bought by Intel in '09
- Available in the Intel Compilers

Work-Stealing

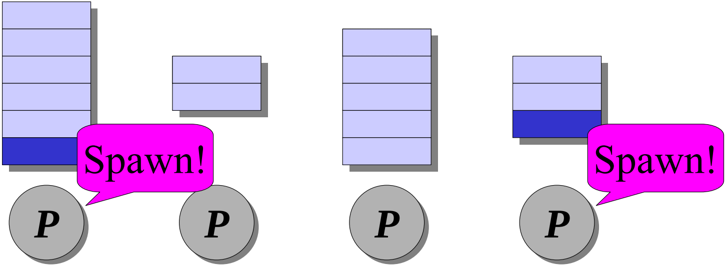
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



With material by
Charles E. Leiserson (MIT)

Work-Stealing

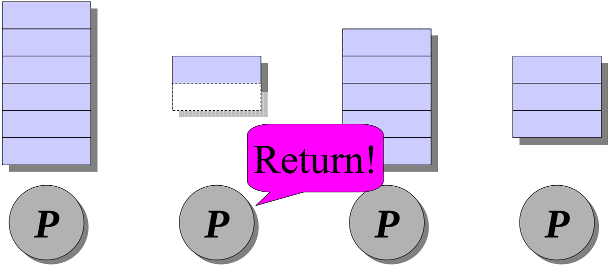
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



With material by
Charles E. Leiserson (MIT)

Work-Stealing

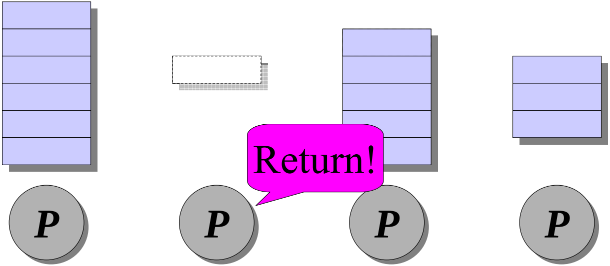
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



With material by
Charles E. Leiserson (MIT)

Work-Stealing

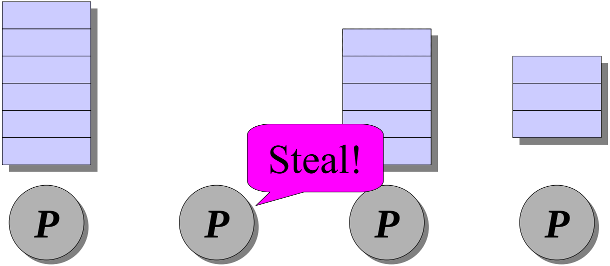
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



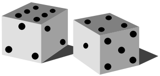
With material by
Charles E. Leiserson (MIT)

Work-Stealing

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



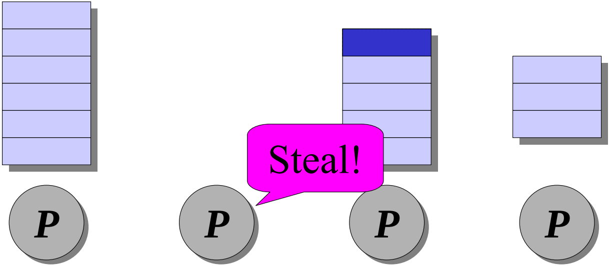
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



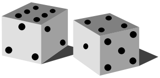
With material by
Charles E. Leiserson (MIT)

Work-Stealing

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



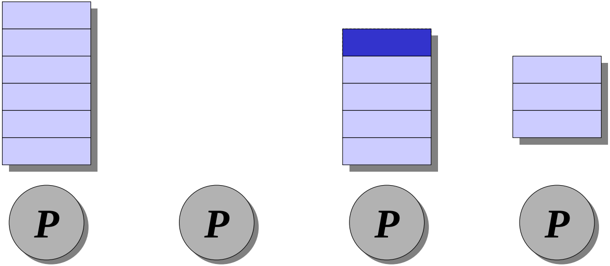
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



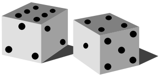
With material by
Charles E. Leiserson (MIT)

Work-Stealing

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



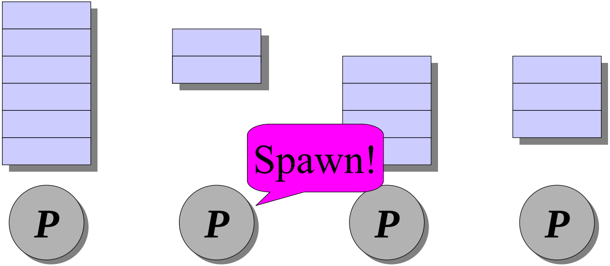
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



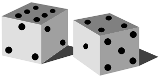
With material by
Charles E. Leiserson (MIT)

Work-Stealing

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



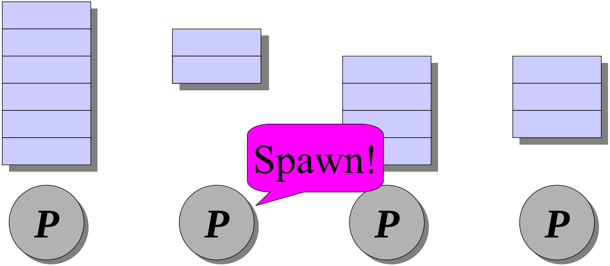
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



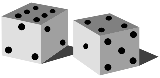
With material by
Charles E. Leiserson (MIT)

Work-Stealing

Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a processor runs out of
rom the
s deque.



Why is Work-Stealing better than a Task Queue?

With material by
Charles E. Leiserson (MIT)

General Graphs: Implementations

- Intel Cilk(+) (also: vector math)
- OpenCL ("Events", Out-of-order queues)
- Intel Thread Building Blocks
- StarPU
- (Charm++)
- Many more

General Graphs: Issues

- Model can accommodate 'speculative execution'
  - Launch many different 'approaches'
  - Abort the others as soon as one satisfactory one emerges.
- Discover dependencies, make up schedule at run-time
  - Usually less efficient than the case of known dependencies
  - Map-Reduce absorbs many cases that would otherwise be general
- On-line scheduling: complicated
- Not a good fit if a more specific pattern applies
- Good if inputs/outputs/functions are (somewhat) heavy-weight

Questions?

?

Image Credits

- Pipe: sxc.hu/mterraza
- Tree: sxc.hu/bertvthul
- Radar: sxc.hu/KimPouss
- Quadtree: flickr.com/ethanhein (cc)