Lecture 10: Parallel Patterns: The What and How of Parallel Programming

G63.2011.002/G22.2945.001 · November 9, 2010

Tentative Plan for Rest of Class

- Today: Parallel Patterns
- Nov 16: Load Balancing
- Nov 23: More performance tricks, tools
- Nov 30: Odds and Ends in GPU Land
- Dec 7: moved to Dec 14 (still ok?)
- Dec 14, 21: Final Project Presentations
  - Will assign presentation date this week.

Tentative Plan for Rest of Class

- Today: Parallel Patterns
- Nov 16: Load Balancing
- Nov 23: More performance tricks, tools
- Nov 30: Odds and Ends in GPU Land
- Dec 7: moved to Dec 14 (still ok?)
- Dec 14, 21: Final Project Presentations
  - Will assign presentation date this week.

Anything not on here that you would like covered?

Tentative Plan for Rest of Class

- Today: Parallel Patterns
- Nov 16: Load Balancing
- Nov 23: More performance tricks, tools
- Nov 30: Odds and Ends in GPU Land
- Dec 7: moved to Dec 14 (still ok?)
- Dec 14, 21: Final Project Presentations
  - Will assign presentation date this week.

Will post HW3 solution soon. (list message)
Graded HW3 next week.

Today

"Traditional" parallel programming in a nutshell

Key question:

- Data Dependencies

Outline

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

Outline

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

Embarrassingly Parallel

$$y_i = f_i(x_i)$$

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

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Embarrassingly Parallel

When does a function have a "side effect"?

In addition to producing a value, it

- modifies non-local state, or
- has an observable interaction with the outside world.

where $i \in \{1$

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Embarrassingly Parallel

$$y_i = f_i(x_i)$$

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

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Embarrassingly Parallel

$$y_i = f_i(x_i)$$

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

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Often: $f_1 = \cdots = f_N$. Then

- Lisp/Python function `map`
- C++ STL `std::transform`

Embarrassingly Parallel: Graph Representation

Embarrassingly Parallel: Graph Representation

Trivial? Often: no.

Embarrassingly Parallel: Examples

Surprisingly useful:

- Element-wise linear algebra: Addition, scalar multiplication (*not* inner product)
- Image Processing: Shift, rotate, clip, scale, ...
- Monte Carlo simulation
- (Brute-force) Optimization
- Random Number Generation
- Encryption, Compression (after blocking)
- Software compilation
  - `make -j8`

Embarrassingly Parallel: Examples

Surprisingly useful:

- Element-wise linear algebra: Addition, scalar multiplication (*not* inner product)
- Image Processing: Shift, rotate, clip, scale, ...
- Monte Carlo simulation
- (Brute-force) Optimization
- Random Number Generation
- Encryption, Compression (after blocking)
- Software compilation
  - `make -j8`

But: Still needs a minimum of coordination. How can that be achieved?

Mother-Child Parallelism

Mother-Child parallelism:

(formerly called "Master-Slave")

Embarrassingly Parallel: Issues

- Process Creation: Dynamic/Static?
  - MPI 2 supports dynamic process creation
- Job Assignment ('Scheduling'): Dynamic/Static?
- Operations/data light- or heavy-weight?
- Variable-size data?
- Load Balancing:
  - Here: easy

Embarrassingly Parallel: Issues

- Process Creation: Dynamic/Static?
  - MPI 2 supports dynamic process creation
- Job Assignment ('Scheduling'): Dynamic/Static?
- Operations/data light- or heavy-weight?
- Variable-size data?
- Load Balancing:
  - Can you think of a load balancing recipe?

Outline

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

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: Graph

Partition: Examples

- Time-marching
  (in particular: PDE solvers)
  - (Including finite differences $\rightarrow$ HW3!)
- Iterative Methods
  - Solve $Ax = b$ (Jacobi, ...)
  - Optimization (all P on single problem)
  - Eigenvalue solvers
- Cellular Automata (Game of Life :-)

Partition: Issues

- Only useful when the computation is mainly local
  - Responsibility for updating one datum rests with one processor
- Synchronization, Deadlock, Livelock, ...
  - Performance Impact
  - Granularity
- Load Balancing: Thorny issue
  - $\rightarrow$ next lecture
- Regularity of the Partition?

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'.

Outline

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

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

Outline

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

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 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.)

Thinking about 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!

Thinking about 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
  - $P > T_1/T_\infty$ c

Algorithm-specific!

Lower Bounds:

- $T_P \geq \ldots ?$ (in terms of T_1)
- $T_P \geq \ldots ?$ (in terms of T_∞)

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 length of the "critical path".
- The first summand can
- Only valid for shared-me

Estimate for $P = 1$?

Proof sketch?

Brent for Reduction

Again: Number of items N.

Brent says

$$T_P = O\left(\frac{T_1}{P} + T_\infty\right) = O\left(\frac{N}{P} + \log N\right).$$

Within a work group: $N = P \Rightarrow T_N = O(\log N)$.

Brent for Reduction

Again: Number of items N.

Brent says

$$T_P = O\left(\frac{T_1}{P} + T_\infty\right) = O\left(\frac{N}{P} + \log N\right).$$

Within a work group: $N = P \Rightarrow T_N = O(\log N)$.

But: Work groups are an illusion! Machine has finite width.
Thus $T_P > O(\log N)$! How low can we take P before we hurt our asymptotic runtime T_P?

Brent for Reduction

Again: Number of items N.

Brent says

$$T_P = O\left(\frac{T_1}{P} + T_\infty\right) = O\left(\frac{N}{P} + \log N\right).$$

Within a work group: $N = P \Rightarrow T_N = O(\log N)$.

But: Work groups are an illusion! Machine has finite width. Thus $T_P > O(\log N)$! How low can we take P before we hurt our asymptotic runtime T_P?

Asymptotically optimal $T_P = O(\log N)$ for

$$P \geq \frac{N}{\log N}.$$

Result: We're free to reduce P by a factor of $(\log N)$ without increasing T_P. $\Rightarrow$ Do $(\log N)$ items in sequence per work item without increasing asymptotic T_P.

Brent for Reduction

Again: Number of iter

Brent says

$$T_P = O\left(\right.$$

Think of this in terms of cost:

$$\text{Cost} = P \times T_P$$

Within a work group: $N = P \Rightarrow T_N = O(\log N)$.

But: Work groups are an illusion! Machine has finite width. Thus $T_P > O(\log N)$! How low can we take P before we hurt our asymptotic runtime T_P?

Asymptotically optimal $T_P = O(\log N)$ for

$$P \geq \frac{N}{\log N}.$$

Result: We're free to reduce P by a factor of $(\log N)$ without increasing T_P. $\Rightarrow$ Do $(\log N)$ items in sequence per work item without increasing asymptotic T_P.

Brent for Reduction

Again: Number of iter

Brent says

$$T_P = O\left(\right.$$

Within a work group:

Think of this in terms of cost:

$$\text{Cost} = P \times T_P$$

Brent gives lower bound on P.

Fewer Processors $\Rightarrow$ less cost!

But: Work groups are an illusion! Machine has finite width. Thus $T_P > O(\log N)$! How low can we take P before we hurt our asymptotic runtime T_P?

Asymptotically optimal $T_P = O(\log N)$ for

$$P \geq \frac{N}{\log N}.$$

Result: We're free to reduce P by a factor of $(\log N)$ without increasing T_P. $\Rightarrow$ Do $(\log N)$ items in sequence per work item without increasing asymptotic T_P.

Brent for Reduction

Again: Number of iter

Brent says

$$T_P = O\left(\right.$$

Within a work group:

But: Work groups are
Thus $T_P > O(\log N)$!
asymptotic runtime T_P?

Think of this in terms of cost:

$$\text{Cost} = P \times T_P$$

Brent gives lower bound on P.

Fewer Processors $\Rightarrow$ less cost!

"*Algorithm cascading*"

Asymptotically optimal $T_P = O(\log N)$ for

$$P \geq \frac{N}{\log N}.$$

Result: We're free to reduce P by a factor of $(\log N)$ without increasing T_P. $\Rightarrow$ Do $(\log N)$ items in sequence per work item without increasing asymptotic T_P.

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

Map-Reduce

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

where N is the input size.

- Lisp naming, again
- Mild generalization of reduction

Map-Reduce: Graph

MapReduce: Discussion

MapReduce $\geq$ `map` + `reduce`:

- Used by Google (and many others) for large-scale data processing
- Map generates (`key`, `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

Embarrassingly Parallel

Partition

Pipelines

Reduction

Scan

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.

- Also called "prefix sum".
- Or cumulative sum ('`cumsum`') by Matlab/NumPy.

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 `Tmp` $=$ `L` $+$ `R` and pass to parent

On downward sweep:

- Get value `Tmp` from parent
- Send `Tmp` to left ch
- 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

Questions?

?