Part V: Algorithms & Data Structs

Goal: Focus more closely on scalable parallel techniques, both computation and data

Reconceptualizing a Computation

- Good parallel solutions result from rethinking a computation …
  - Sometimes that amounts to reordering scalar operations
  - Sometimes it requires starting from scratch
- The SUMMA matrix multiplication algorithm is the poster computation for rethinking!

This computation is part of homework assignment
Return To A Lecture 1 Computation

Matrix Multiplication on Processor Grid

- Matrices $A$ and $B$ producing $n \times n$ result $C$ where
  \[
  C_{rs} = \sum_{1 \leq k \leq n} A_{rk} B_{ks}
  \]

Applying Scalable Techniques

- Assume each processor stores block of $C$, $A$, $B$; assume “can’t” store all of any matrix
- To compute $c_{rs}$ a processor needs all of row $r$ of $A$ and column $s$ of $B$
- Consider strategies for minimizing data movement, because that is the greatest cost -- what are they?
Grab All Rows/Columns At Once

- If all rows/columns are present, it's local

![Diagram of matrix multiplication]

- Each element requires $O(n)$ operations
- Modern pipelined processors benefit from large blocks of work
- But memory space and BW are issues

Process $t \times t$ Blocks

- Referring to local storage

```c
for (r=0; r < t; r++){
    for (s=0; s < t; s++){
        c[r][s] = 0.0;
        for (k=0; k < n; k++){
            c[r][s] += a[r][k]*b[k][s];
        }
    }
}
```

![Diagram of matrix multiplication]

- Sweeter caching
- Only move a $t \times t$ block at a time
Change Of View Point

- Don’t think of row-times-column

Switch orientation -- by using a column of A and a row of B compute all 1st terms of the dot products

SUMMA

- Scalable Universal Matrix Multiplication Alg
  - Invented by van de Geijn & Watts of UT Austin
  - Claimed best machine independent MM

- Whereas MM is usually A row x B column, SUMMA is A column x B row because computation switches sense
  - Normal: Compute all terms of dot product
  - SUMMA: Compute first term of all dot products

Strange. But fast!
SUMMA Assumptions

- Threads have two indices, handle \( t \times t \) block
- Let \( p = P^{1/2} \), then thread \( u,v \)
  - reads all columns of \( A \) for indices \( u*t:(u+1)*t-1,j \)
  - reads all rows of \( B \) for indices \( i,v*t:(v+1)*t-1 \)
  - The arrays will be in “global” memory and referenced as needed

Higher Level SUMMA View

- See SUMMA as an iteration multicasting columns and rows
- Each processor is responsible for sending/recieving its column/row portion at proper time
- Followed by a step of computing next term locally

[Diagram of SUMMA assumptions and higher level view]

[Link: www.cs.utexas.edu/users/rvdg/abstracts/SUMMA.html]
Summary of SUMMA

- **Facts:**
  - vdG & W advocate blocking for msg passing
  - Works for $A$ being $m \times n$ and $B$ being $n \times p$
  - Works fine when local region is not square
  - Load is balanced esp. of Ceiling/Floor is used
  - Fastest machine independent MM algorithm!

- **Key algorithm for 524:** Reconceptualizes MM to handle high $\lambda$, balance work, use BW well, exploit efficiencies like multicast, …

Schwartz’s Algorithm

- Jack Schwartz (NYU) asked: What is optimal number of processors to combine $n$ values?
  - Reasonable Answer: binary tree w/ values at leaves has $O(\log n)$ complexity
  - To this solution add $\log n$ values into each leaf
  - Same complexity ($O(\log n)$), but $n\log n$ values!
  - Asymptotically, the advantage is small, but the tree edges require communication
**Schwartz’ Algorithm**

- Jack Schwartz (NYU) asked: What is optimal number of processors to combine $n$ values?
  - Reasonable Answer: binary tree w/ values at leaves has $O(\log n)$ complexity
  - To this solution add $\log n$ values into each leaf
  - Same complexity ($O(\log n)$), but $n\log n$ values!
  - Asymptotically, the advantage is small, but the tree edges require communication

**Schwartz**

- Generally $P$ is not a variable, and $P << n$
- Use **Schwartz as heuristic**: Prefer to work at leaves (no matter how much smaller $n$ is that $P$) rather than enlarge (make a deeper) tree, implying tree will have no more than $\log_2 P$ height
- Also, consider higher degree tree -- in cases of parallel communication (CTA) some of the communication may overlap
Block Allocations

- The Red/Blue computation illustrated a 2D-block data parallel allocation of the problem.
- Generally block allocations are better for data transmission: surface to volume advantage ... since only edges are x-mitted.

![Diagram showing block data parallel allocation]

Now scale problem 4x.

Different Regimens

- Though block is generally a good allocation, it's not absolute:
  - P=1, all comm wasted
  - P=2, row-wise saves column comm
  - P=4, rows and blocks are a wash
  - Where is the point of dim. return?

![Diagram comparing different regimens]
Shadow Regions/Fluff

- To simplify local computation in cases where nearest neighbor’s values transmitted, allocate in-place memory to store values:

- Array can be referenced as if it’s all local

Aspect Ratio

- Generally $P$ and $n$ do not allow for a perfectly balanced allocation …

- Several ways to assign arrays to processors

- 13x13 on 4x4 process array

- Quotient + remainder

- Ceiling + floor

- Generally a small effect
Assigning Processor 0 Work

- $p_0$ is often assigned “other duties”, such as:
  - Orchestrate I/O
  - Root node for combining trees
  - Work Queue Manager …

- Assigning $p_0$ the smallest quantum of work helps it avoid becoming a bottleneck
  - For either quotient + remainder or ceiling/floor $p_0$ should be the last processor

Locality Always Matters

- Array computations on CMPs
  - Dense Allocation vs Fluff
  - Issue is cache invalidation
  - Keeping MM managed intermediate buffers keeps array and fluff local (L1)
  - Sharing causes elements at edge to repeatedly invalidate harming locality

False sharing an issue, too
Load Balancing

- Certain computations are inherently imbalanced ... LU Decomposition is one
- Standard block decomposition quickly becomes very biased

Cyclic & Block Cyclic

- Cyclic allocation means “to deal” the elements to the processes like cards
- Allocating 64 elements to five processes: black, white, three shades of gray
- Block cyclic is the same idea, but rather with regular shaped blocks
Block Cyclic

- Consider the LU matrix allocated in 3x2 blocks to four processes:
- The check it midway in the computation

Opportunities To Apply Cyclic

- The technique applies to work allocation as well as memory allocation

Julia Set from http://aleph0.clarku.edu/~djoyce/
Announcements

☐ No class Monday
☐ Assignment at end of class today
☐ Make-up Class Friday Feb. 29th

Generalized Reduce and Scan

☐ The importance of reduce/scan has been repeated so often, it is by now our mantra
☐ In nearly all languages the only available operators are $+, \times, \text{min}, \text{max}, \&\&, ||$
☐ The concepts apply much more broadly
☐ Goal: Understand how to make user-defined variants of reduce/scan specialized to specific situations

--- Seemingly sequential looping code can be UD-scan ---
Examples Applicable Computations

- **Reduce**
  - Second smallest, or generally, kth smallest
  - Histogram, counts items in k buckets
  - Length of longest run of value 1s
  - Index of first occurrence of x

- **Scan**
  - Team standings
  - Find the longest sequence of 1s
  - Index of most recent occurrence

---

Structure of Computation

- **Begin by applying Schwartz idea to problem**
  - Local computation
  - Global $\log P$ tree

---

More computation at nodes is OK
Introduce Four Functions

- Make four non-communication operations
  - `init()` initialize the reduce/scan
  - `accum()` perform local computation
  - `combine()` perform tree combining
  - `x_gen()` produce the final result for either op
    - `x = reduce`
    - `x = scan`

- Incorporate into Schwartz-type logic

  Think of: `reduce(fi, fa, fc, fg)`

Assignment of Functions

- Init: Each leaf
- Accum: Aggregate each array value
- Combine: Each tree node
- `reduceGen`: Root
Example: +<<A Definitions

- Sum reduce uses a temporary value, called a tally, to hold items during processing.
- Four reduce functions:
  - `tally init()` { `tal = new tally; tal=0; return tal;` }
  - `tally accum(int op_val, tally tal)`
    { `tal += op_val; return tal;` }
  - `tally combine(tally left, tally right)`
    { `return left + right;` }
  - `int reduce_gen(tally ans)` { `return ans;` }

More Involved Case

- Consider Second Smallest -- useful, perhaps for finding smallest nonzero among non-negative values.
- `tally` is a struct of the smallest and next smallest found so far `{float sm, nsm}`
- Four functions:
  - `tally init()` {
    `pair = new tally;`
    `pair.sm = maxFloat;`
    `pair.nsm = maxFloat;`
    `return pair;` }

31

32
Second Smallest (Continued)

- **Accumulate**

  ```c
  tally accum(float op_val, tally tal) {
    if (op_val < tal.sm) {
      tal.nsm = tal.sm;
      tal.sm = op_val;
    } else {
      if (op_val > tal.sm && op_val < tal.nsm)
        tal.nsm = op_val;
    }
    return tal;
  }
  ```

  Finds 2nd smallest distinct value

- **Combine**

  ```c
  tally combine(tally left, tally right){
    accum(left.sm, right);
    accum(left.nsm, right);
    return right;
  }
  ```

  ```c
  int reduce_gen(tally ans) {return ans.nsm;}
  ```

  - Notice that the signatures are all different
  - Conceptually easy to write equivalent code, but reduction abstraction clarifies
Recall Parallel Prefix Algorithm

Compute sum going up
Figure prefixes going down

Invariant: Parent data is prefix of elements to left of subtree

DG's ML Parallel Prefix I

(* e1 $$ e2 means "do e1 and e2 in parallel and return the two results as a pair." But this is a placeholder -- rather than create Threads, I just compute e1 and e2 sequentially. The point is $$ marks where one _should_ interpose parallelism.*)

let ($$) f1 f2 =
  let x = f1() in
  let y = f2() in
  (x,y)

(* used as intermediate data structure by parallel prefix *)

let rootval tr =
  match tr with
    | Leaf (i,_) -> i
    | Node(i,_,_) -> i
DG’s ML Parallel Prefix II

(* a rather polymorphic parallel prefix pattern *)
let parallelprefix f_up f_leaf f_down down_start arr =
  let ans = Array.create (Array.length arr) down_start in
  let rec firstpass left right =
    if left = right
    then Leaf (f_leaf arr.(left), left) (* the index is useful in secondpass!*)
    else
      let mid = left + (right - left) / 2 in
      let l, r = (fun () -> firstpass left mid)
      $$ (fun () -> firstpass (mid + 1) right) in
      Node(f_up (rootval l) (rootval r), l, r) in
    let rec secondpass fromleft tr =
      match tr with
      Leaf(i, ind) -> ans.(ind) <- f_down fromleft i
      | Node(_, l, r) ->
        ignore((fun () -> secondpass fromleft l))
        $$ (fun () -> secondpass (f_down fromleft (rootval l)) r)) in
      secondpass down_start (firstpass 0 (Array.length arr - 1));
    ans

DG’s ML Parallel Prefix III

(* three example uses, notice in the last one the type changes between input and output *)
let prefix_sum = parallelprefix (+) (fun x -> x) (+) 0

(* only works for nonnegative numbers; easy to fix with an option *)
(* max happens to be in the core library, but of course it is easy to redefine as (fun x y -> if x > y then x else y) *)
let prefix_max = parallelprefix max (fun x -> x) max (-1)

let prefix_string = parallelprefix (^) string_of_int (^) ""
User-Defined Scan

- Consider operations after the reduce is over
- Consider where functions used: i, a, c, sg

The basic scan logic applies functions

Scan Process Assignment

- Init: At root
- Combine: Each interior leaf
- scanGen: Each operand value
Index of Last Occurrence of x

- Assume 0-origin indexing
- `tally` is simply an integer

```c
int tally_init() { idx = new tally;  
    tally idx = -1;  
    return idx;  
}
```

```c
tally accum(int op_val, tally tal, int x, idx) {  
    if (op_val == x)  
        tal = idx;  
    return tal;  
}
```

Last Index (Continued)

```c
tally combine(tally left, tally right) {  
    if (left > right)  
        return left;  
    else  
        return right;  
}
```

```c
int scan_gen(int op_val, tally ans, int x, idx) {  
    if (op_val == x){  
        return idx;  
    } else  
        return ans;  
}
```
Example x == 1

UD-Scan Summary

- User-defined scan extends UD-reduce
- The operations are essentially the same
  - Applied in additional places
  - Applied with additional arguments
- UD-scan is efficient and powerful … if the language you’re writing in doesn’t have it, define your own

To think of scanning takes practice
Applying UD Reduce/Scan

- Discuss computations that can use UD R/S
  - Sample -- The bounding box of a set of points (x,y) in $E_2$ is easy with 4 reduces; do it in 1
  - ...

More Generally: UD-Vector Ops

- Scan maintains “context” allowing ordered operations, but that is often not needed
- Vector operations focus on performing some operation across the elements that has global meaning -- longest run of 1s
  - Like all ||ism, the key is formulating local computation so it can be combined to achieve a global result
  - The “scan driver” probably suffices

Blelloch: Introduced Vector Model for || programming
Tree Algorithms

- Trees are an important component of computing
  - The “Schwartz tree” has been logical
  - Trees as data structures are complicated because they are typically more dynamic
  - Pointers are generally not available
  - Work well with work queue approach
  - As usual, we try to exploit locality and minimize communication

Breadth-first Trees

- Common in games, searching, etc
- Split: Pass 1/2 to other processor, continue
  - Stop when processors exhausted
  - Responsible for tree that remains
  - Ideal when work is localized
**Depth-first**

- Common in graph algorithms

- Get descendants, take one and assign others to the task queue

  Key issue is managing the algorithm’s progress

**Coordination Among Nodes**

- Tree algorithms often need to know how others are progressing
  - Interrupt works if it is just a search: Eureka!!
  - Record $\alpha$-$\beta$ cut-offs in global variable
  - Other pruning data, e.g. best so far, also global
  - Classic error is to consult global too frequently

- Rethink: What is tree data structure’s role?

  Write essay: Dijkstra’s algorithm is not a good… :)

Complications

- If coordination becomes too involved, consider alternate strategies:
  Graph traverse $\Rightarrow$ local traverse of partitioned graph

- Local computation uses sequential tree algorithms directly … stitch together

Full Enumeration

- Trees are a useful data structure for recording spatial relationships: K-D trees

- Generally, decomposition is unnecessary “all the way down” -- but this optimization implies two different regimes
Cap Reduces Communication

- The nodes near root can be stored redundantly

Each process keeps copy of “cap” nodes

- Processors consult local copy -- alter others to changes

Summary of Parallel Algorithms

- Reconceptualizing is often most effective
- Focus has not be on ||ism, but on other stuff
  - Exploiting locality
  - Balancing work
  - Reducing inter-thread dependences
- We produced general purpose solution mechanisms: UD-reduce and UD-scan
- We like trees, but recognize that direct application is not likely
Assignment 6

- For next Wednesday (2/20): Write an MPI program for the SUMMA alg
  - Create rectangular arrays A, B, C, filling A, B
  - Send portions of A, B to worker processes
  - Iterate over common dimension,
    - send columns of A, rows of B to other processes
    - for each, multiply A elements times B elements and accumulate into local portion of C
  - Measure time, except for initialization, and report the “usual stuff” for different numbers of processes