Architecture and Parallel Computers

Chris Kauffman

CS 499: Spring 2016 GMU
Logistics

Reading: Grama Ch 2 + 3

- Ch 2.3-5 is most important for Ch 2
- Ch 3 all

Assignment 1

- Posted, due Thu 2/4
- Groups of 2 permitted
- Questions?
- Office hours Tue 3:30-5:30pm

Today

- Finish Parallel architecture (HW1: #1-2)
- Parallel Algorithm Decomposition (HW1: #3,4,6)
Dependency Graphs

- Relation of tasks to one another
- Vertices: tasks, often labeled with time to complete
- Edges: indicate what must happen first
- Should be a DAG, Directed Acyclic Graph
  (If not, you’re in trouble)
Features of Dependency Graphs

- Critical Path Length = Sum of longest path
- Maximum Degree of Concurrency = # of task in "widest" section
- Average Degree of Concurrency = \[
\frac{\text{Sum of all vertices}}{\text{Critical Path Length}}
\]
Computing Features of Dependency Graphs

Maximum Degree of Concurrency

- (a) 4
- (b) 4

Total Task Work

- (a) 63
- (b) 64

Critical Path Length

- (a) 27 (leftmost path)
- (b) 34 (rightmost)

Average Degree of Concurrency

- (a) \( \frac{63}{27} = 2.33 \)
- (b) \( \frac{64}{34} = 1.88 \)
Exercise: Compute Features of Dependency Graph

Compute

- Total Work
- Maximum degree of concurrency
- Critical Path Length
- Average Degree of Concurrency
Most build systems for programs calculate task graphs. Makefiles describe DAGs to build projects with `make`.

```
count_words: count_words.o lexer.o
  gcc count_words.o lexer.o -lfl \ 
  -o count_words

count_words.o: count_words.c
  gcc -c count_words.c

lexer.o: lexer.c
  gcc -c lexer.c

lexer.c: lexer.l
  flex -t lexer.l > lexer.c

.PHONY: clean
clean:
  rm -rf *.o lexer.c count_words
```

Look up `make -j 4` option: use 4 processors for concurrency.
Identifying Tasks for Parallel Programs

- This is the tricky part
- Several techniques surveyed in the text that we’ll overview
- Two general paradigms for creating parallel programs

Parallelize a Serial Code
- Already have a solution to the problem
- Identify tasks within solution
- Construct a task graph and parallelize based on it
- We’ll spend most of our time on this as it is more common

Redesign for Parallelism
- Best serial code may not parallelize well
- Change the approach entirely to exploit parallelism
- Usually harder, more special purpose, spend less time on it
Recursion Provides Parallelism

Algorithms which use *multiple* recursive calls provide easy opportunities for parallelism

**Multiple Recursive Call Algs**

- Fibonacci calculations
- Mergesort
- Quicksort
- Graph searches

All reasonable for parallelizing: recursive calls are independent, represent independent tasks which can be run in parallel: parallelize a serial alg

Source: Wikipedia Mergesort
Reformulation As Recursive Algorithms

- Can sometimes reformulate an iterative algorithm as a recursive one: Redesign for parallelism
- Show task graph for RECURSIVE_MIN on array

\[ A = \{4, 9, 1, 7, 8, 11, 2, 12\}\]
\[ n = 8 \]

```plaintext
procedure SERIAL_MIN (A, n) begin
min = A[0];
for i := 1 to n - 1 do
   if (A[i] < min) then
      min := A[i];
   endif
endfor;
return min;
end SERIAL_MIN
```

```plaintext
procedure RECURSIVE_MIN (A, n) begin
if (n = 1) then
   min := A[0];
else
   lmin := RECURSIVE_MIN (A, n/2);
   rmin := RECURSIVE_MIN &(A[n/2]), n - n/2);
   if (lmin < rmin) then
      min := lmin;
   else
      min := rmin;
   endelse;
endelse;
return min;
end RECURSIVE_MIN
```
## Data Decomposition: this is the big one

### Output Partitioning
- Collection of output data
- Tasks to compute output data are (relatively) independent
- Parallelize by assigning tasks based on output

### Input Partitioning
- Output tasks not easily independent
- Can build up output via independent tasks on input
- Requires a way to combine results from different sections of input
- Parallelize by assigning tasks to chunks of input then combining

- Combinations of Input/Output partitioning are common
- Examples to follow
Matrix-Vector Multiplication

- Input: matrix $A$, vector $x$
- Output: vector $b$

$$A \times x = b$$

Output Partitioning

- Task to compute each element of output $b$
- Each processor holds rows of $A$ and all of $x$

Input Partitioning

- **Constraint:** Processors have little memory, can’t hold whole rows of $A$ and all of $x$
- **Discuss** an input partitioning: chunks of $A$ and $x$, do some computation, combine results to form elements of $b$
Input Partitioning for Matrix-Vector Multiplication

- Most Tasks: multiply part of a row of $A$ with part of $x$
- Some Tasks: combine partial sums to produce single element of output $b$
Exercise: Frequent Item Set Calculation

Typical data mining task: count how many times items \{D, E\} were bought together in a database of transactions

- **Input:** database + itemsets of interest
- **Output:** frequency of itemsets of interest

Describe tasks for...

- Input partitioning
- Output partitioning
- Combined partitioning

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, E, G, H</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>3</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, E</td>
<td>2</td>
</tr>
<tr>
<td>F, G, H, K,</td>
<td>C, D</td>
<td>1</td>
</tr>
<tr>
<td>A, E, F, K, L</td>
<td>D, K</td>
<td>2</td>
</tr>
<tr>
<td>G, H, L</td>
<td>D, E, F, K, L</td>
<td>1</td>
</tr>
<tr>
<td>F, G, H, L</td>
<td>C, D, K</td>
<td>0</td>
</tr>
</tbody>
</table>

Answers in Grama 3.2
Exploratory Decomposition

Problem Formulations

- Graph Breadth-first and depth-first search
- Path finding in discrete environments
- Combinatorial search (15-puzzle)
- Find a good move in a game (Chess, Go)

Algorithms

- Similar to recursive decomposition
- Each step has several possibilities to explore
- Serial algorithm must try one, then unwind
- Parallel algorithm may explore multiple paths simultaneously
Features of Exploratory Decomposition

- Data duplication may be necessary so each PE can change its own data (puzzle)
- Redundancy may occur: two PEs arrive at the same puzzle state
  - Detect duplication requires programming/communication
  - Ignoring duplication wastes PE time
- Termination is trickier: once a solution is found, must signal to all active PEs that they can quite or move on
- Can lead to strange "super-linear" speedups over serial algorithms or to much wasted effort
Static and Dynamic Task Generation

Static Task Generation

- All tasks known ahead of time
- Easier to plan and distribute data
- Examples abound: matrix operations, sorting (mostly), data analysis, image processing

Dynamic Task Generation

- Tasks are "discovered" during the program run
- Tougher to deal with scheduling, data distribution, coordination
- Difficulty with message passing paradigm
- Examples: game tree search, some recursive algorithms, others(?)

Focus on Static Task Generation
Static and Dynamic Scheduling (Mapping)

- Given tasks and dependencies, must schedule them to run on actual processors
- Problems to solve include Load imbalance (unequal work), Communication overhead, Data distribution as work changes

Static Mapping/Scheduling

- Specify which tasks happen on which processes ahead of time
- Usually baked into the code/algorithm
- Works well for message passing/distributed paradigm

Dynamic Mapping/Scheduling

- Figure out where tasks get run as you go
- More or less required if tasks are "discovered"
- Centralized scheduling Schemes: manager tracks tasks in a data structure, doles out to workers
- Distributed scheduling schemes: workers share tasks directly
Parallel algorithms always introduce overhead: work that doesn’t exist in a serial computation. Reducing overhead usually comes in three flavors.

1. Make tasks as independent as possible
2. Minimize data transfers
3. Overlap communication with computation

#1 and #2 are often in tension: why?
Broad Categories of Parallel Program Designs

Data-parallel
Every processors gets data, computes similar things, syncs data with group, repeats; Example: matrix multiplication

Task Graph
Every processor gets some tasks and associated data, computes then syncs, Example: parallel quicksort (later)

Work-pool and Manager/Workers
Initial tasks go into pool, doled out to workers, discover new tasks, go into pool, distributed to workers.... Example: web server

Stream / Pipeline / Map-Reduce
Raw data goes in, comp1 done to it, fed to comp2, then to comp3, etc. Example: Frequency counts of all documents, LU factorization
Exercise: HW1’s Heat Problem

What are the tasks? How does the task graph look?
What kind of scheduling seems like it will work?
How should the data be distributed?
What broad category of approach seems to fit?

Data parallel, Task graph distribution, Work-pool/Manager-worker, Stream/Pipeline