Principles of Parallel Algorithm Design

Chris Kauffman

Last Updated:
Sun Sep 12 10:33:46 PM CDT 2021

Logistics

Reading: Grama Ch 2 + 3

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

Assignment 1

- Office Hours Tue/Thu 2-2:30pm
- **Questions?**

This Week

- 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

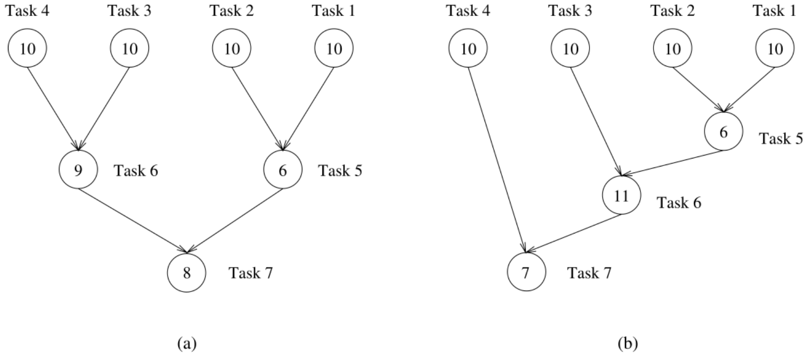


Figure 3.5 Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

- 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

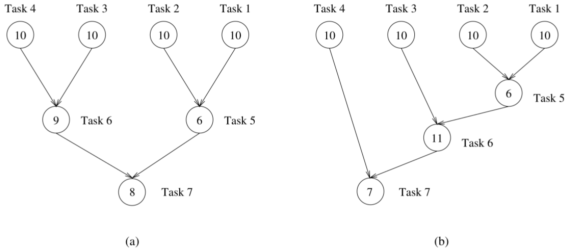


Figure 3.5 Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

Critical Path Length

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

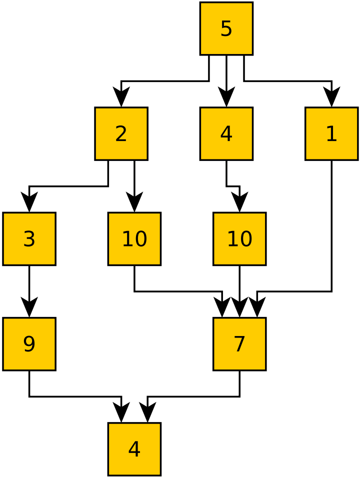
Average Degree of Concurrency

- (a) $63 / 27 = 2.33$
- (b) $64 / 34 = 1.88$

Exercise: Compute Features of Dependency Graph

Compute

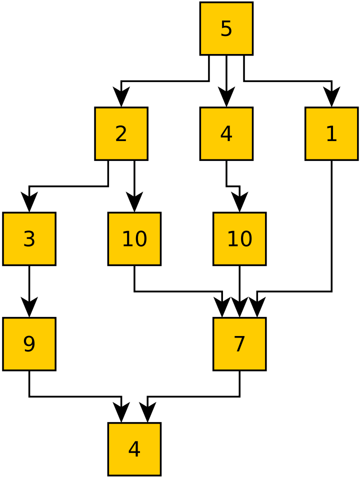
- Total Work
- Maximum degree of concurrency
- Critical Path Length
- Average Degree of Concurrency



Answers: Compute Features of Dependency Graph

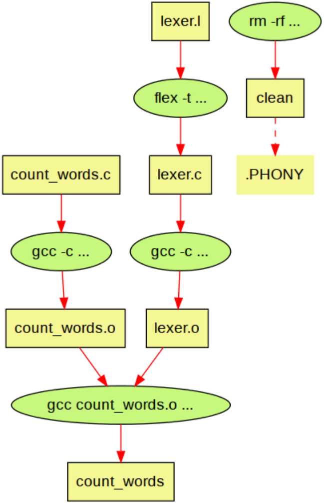
Compute

- Total Work: 55
- Maximum deg of concur.: 3
- Critical Path Length: 30
- Average Deg. of Concur.: $55/30 = 1.83$



Makefiles

- Most build systems for programs calculate task graphs
- `Makefiles` describe DAGs to build projects with `make`



Source: Luke Luo

```
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, we will spend less time on it

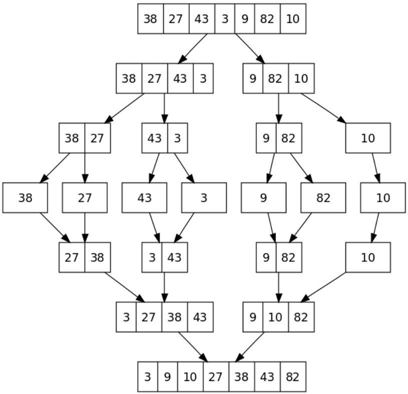
Recursion Provides Parallelism

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

Multiple Recursive Call Algs

- Fibonacci calculations
- Mergesort
- Quicksort
- Graph searches

All allow for parallelizing: recursive calls are independent, represent independent tasks which can be run in parallel BUT not all provide practical benefit when run in parallel



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

```
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
```

```
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: the GOTO Design Technique

Output Partitioning

- Among algorithm Output Data...
- Determine if tasks to compute output are (relatively) independent
- Parallelize by assigning tasks to Procs based on Output that will be on the Proc

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

- Many examples to follow

Exercise: Matrix-Vector Multiplication

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

```
A * x = b
```

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

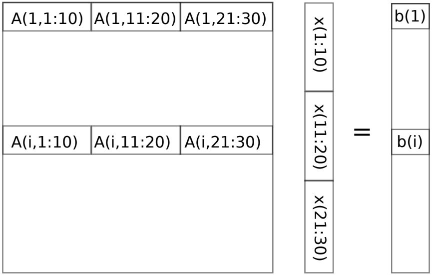
Output Partitioning

- 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`
- Propose an input partitioning: chunks of `A` and `x`, do some computation, combine results to form elements of `b`

Answers: Input Partitioning for Matrix-Vector Multiplication



Task 1: tmp(1,1) = A(1,1:10)*x(1:10)
Task 2: tmp(1,2) = A(1,11:20)*x(11:20)
Task 3: tmp(1,3) = A(1,21:30)*x(21:30)
Task 4: b(1) = tmp(1,1) + tmp(1,2) + tmp(1,3)
...
Task 4*i+1: tmp(i,1) = A(1,1:10)*x(1:10)
Task 4*i+2: tmp(i,2) = A(1,11:20)*x(11:20)
Task 4*i+3: tmp(i,3) = A(1,21:30)*x(21:30)
Task 4*i+4: b(i) = tmp(i,1) + tmp(i,2) + tmp(i,3)

- 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`

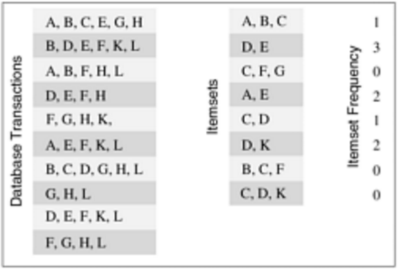
Exercise: Frequent Item Set Calculation

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



Answers: Frequent Item Set Calculation

Output Partitioning

- Whole Database fits on each Proc
- Divide up Itemsets among Procs
- Each Proc scans whole DB counting its Itemsets

Input Partitioning

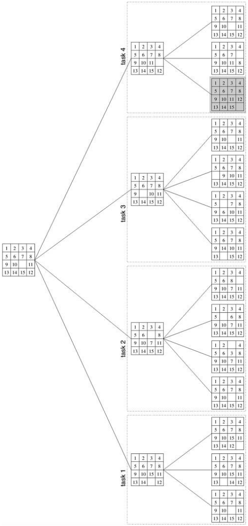
- DB spread across Procs, each has Partial DB
- Assume each Proc can hold all Itemsets
- Each Proc scans its DB portion, counts all Itemsets
- Procs communicate to Sum all itemsets (Reduction)

Combined Partitioning

- DB and Itemsets Spread Across Procs
- Follow Input Partitioning except...
- Procs only communicate in Groups based on Itemsets

More Details in Grama 3.2

Exploratory Decomposition



Problem Formulations

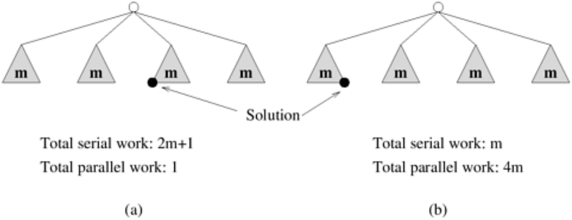
- Graph Breadth-first and depth-first search
- Path finding in discrete environments
- Combinatorial search (15-puzzle)
- Find a good move in a game (Chess, Go)

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 state)
- Redundancy may occur: two PEs arrive at the same 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

We will 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

Reducing the Overhead of Parallelism

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)

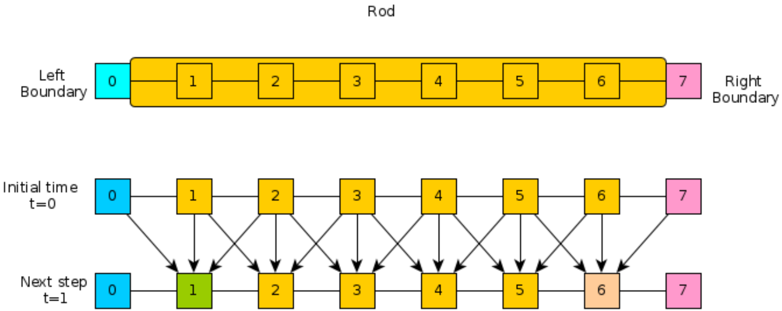
Work-pool and Manager/Workers

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: A1'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

Answers: HW1's Heat Problem

Well, it wouldn't be much of an assignment if I gave you my answers...