CSE 332: Analysis of Fork-Join Parallel Programs

Richard Anderson, Steve Seitz
Winter 2014
Parallel Sum

- Sum up N numbers in an array
Parallel Max?
Reductions

• Same trick works for many tasks, e.g., \((a + b) + c = a + (b + c)\)
  – is there an element satisfying some property (e.g., prime)
  – left-most element satisfying some property (e.g., first prime)
  – smallest rectangle encompassing a set of points (proj3)
  – counts: number of strings that start with a vowel
  – are these elements in sorted order?

• Called a reduction, or reduce operation
  – reduce a collection of data items to a single item
    • result can be more than a single value, e.g., produce histogram from a set of test scores

• Very common parallel programming pattern
Parallel Vector Scaling

- Multiply every element in the array by 2

N proc (threads) multiply each separately then chart
Maps

• A map operates on each element of a collection of data to produce a new collection of the same size
  – each element is processed independently of the others, e.g.
    • vector scaling
    • vector addition
    • test property of each element (is it prime)
    • uppercase to lowercase
    • ...

• Another common parallel programming pattern
Maps in ForkJoin Framework

class VecAdd extends RecursiveAction {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) { ... }
    protected void compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            for (int i = lo; i < hi; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (hi + lo) / 2;
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);
            left.fork();
            right.compute();
            left.join();
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    fjPool.invoke(new VecAdd(0, arr.length, ans, arr1, arr2);
    return ans;
}
Maps and Reductions

Maps and reductions: the “workhorses” of parallel programming

– By far the most important and common patterns

– Learn to recognize when an algorithm can be written in terms of maps and reductions

– makes parallel programming easy (plug and play)
Distributed Map Reduce

• You may have heard of Google’s map/reduce
  – or open-source version called Hadoop
  – powers much of Google’s infrastructure

• Idea: maps/reductions using many machines
  – same principles, applied to distributed computing
  – system takes care of distributing data, fault-tolerance
  – you just write code to handle one element, reduce a collection

• Co-developed by Jeff Dean (UW alum!)
Maps and Reductions on Trees

- Max value in a min-heap

- How to parallelize?
- Is this a map or a reduce?
- Complexity? $O(\log N)$
Analyzing Parallel Programs

Let $T_p$ be the running time on $P$ processors

Two key measures of run-time:

- **Work**: How long it would take 1 processor = $T_1$
- **Span**: How long it would take infinity processors = $T_\infty$
  - The hypothetical ideal for parallelization
  - This is the longest “dependence chain” in the computation
  - Example: $O(\log n)$ for summing an array
  - Also called “critical path length” or “computational depth”
The DAG

• Fork-join programs can be modeled with a DAG
  – nodes: pieces of work
  – edges: order dependencies

A fork creates two children
  • new thread
  • continuation of current thread

A join makes a node with two incoming edges
  • terminated thread
  • continuation of current thread

What’s $T_1$ (work): sum of all nodes

What’s $T_\infty$ (span): longest path
Divide and Conquer Algorithms

Our fork and join frequently look like this:

In this context, the span \((T_\infty)\) is:

- The longest dependence-chain; longest ‘branch’ in parallel ‘tree’
- Example: \(O(\log n)\) for summing an array; we halve the data down to our cut-off, then add back together; \(O(\log n)\) steps, \(O(1)\) time for each
- Also called “critical path length” or “computational depth”
Parallel Speed-up

• Speed-up on $P$ processors: $\frac{T_1}{T_P}$

• If speed-up is $P$, we call it perfect linear speed-up
  – e.g., doubling $P$ halves running time
  – hard to achieve in practice

• Parallelism is the maximum possible speed-up: $\frac{T_1}{T_\infty}$
  – if you had infinite processors
Estimating $T_p$

- How to estimate $T_p$ (e.g., $P = 4$)?

- Lower bounds on $T_p$ (ignoring memory, caching...)
  1. $T_\infty$
  2. $T_1 / P$
     - which one is the tighter (higher) lower bound?

- The ForkJoin Java Framework achieves the following expected time asymptotic bound:
  $$T_p \in O(T_\infty + T_1 / P)$$
  - this bound is optimal
Amdahl’s Law

• Most programs have
  1. parts that parallelize well
  2. parts that don’t parallelize at all

• The latter become bottlenecks
Amdahl’s Law

• Let $T_1 = 1$ unit of time
• Let $S =$ proportion that can’t be parallelized

$$1 = T_1 = S + (1 - S)$$

• Suppose we get perfect linear speedup on the parallel portion:

$$T_P =$$

• So the overall speed-up on $P$ processors is (Amdahl’s Law):

$$T_1 / T_P =$$

$$T_1 / T_\infty =$$

• If $1/3$ of your program is parallelizable, max speedup is:
• Suppose 25% of your program is sequential.
  – Then a billion processors won’t give you more than a 4x speedup!

• What portion of your program must be parallelizable to get 10x speedup on a 1000 core GPU?
  – $10 \leq \frac{1}{S + (1-S)/1000}$

• Motivates minimizing sequential portions of your programs
Take Aways

• Parallel algorithms can be a big win

• Many fit standard patterns that are easy to implement

• Can’t just rely on more processors to make things faster (Amdahl’s Law)