Computational Optimization
ISE 407

Lecture 12

Dr. Ted Ralphs

Reading for This Lecture

- "Introduction to High Performance Computing", V. Eijkhout, Chapter 2.
- "Introduction to High Performance Computing for Scientists and Engineers," G. Hager and G. Wellein, Chapter 5.
- Assessing the Effectiveness of (Parallel) Branch-and-Bound Algorithms
- Paper by Kumar and Gupta
- Paper by Gustafson
- Roosta, Chapter 5

Empirical Analysis of Parallel Algorithms

- Modern parallel computing platforms are essentially all asynchronous.
- Threads/processes do not share a global clock.
- In practice, this means that the execution of parallel algorithms is non-deterministic (although specific implemenations can overcome this).
- This means that theoretical analysis of complex parallel algorithms designed to be run on asynchronous distributed hardware is exceedingly difficult.
- For analysis of all but the simplest parallel algorithms, we must depend primarily on empirical analysis.
- The realities ignored by our models of parallel computation are actually important in practice.

What are the Goals?

- *Sequential efficiency*: Amount of one variable resource (typically time) required for a sequential algorithm to perform a fixed computation.
- *Parallel scalability*: Measures tradeoff between resources.
  - Classical (Strong): Amount of one resource required for a parallel system to perform a fixed computation as a function of other resources (cores).
  - Classical (Weak): Amount of computation that can be done in fixed wallclock time as a function of system resources.
  - Alternative (Weak): Amount of computation that can be done with fixed total resources as a function of wallclock time.
- *Overall "Effectiveness"*: Resources required to perform a fixed computation on a parallel system with fixed resources.

(Strong) Scalability Example

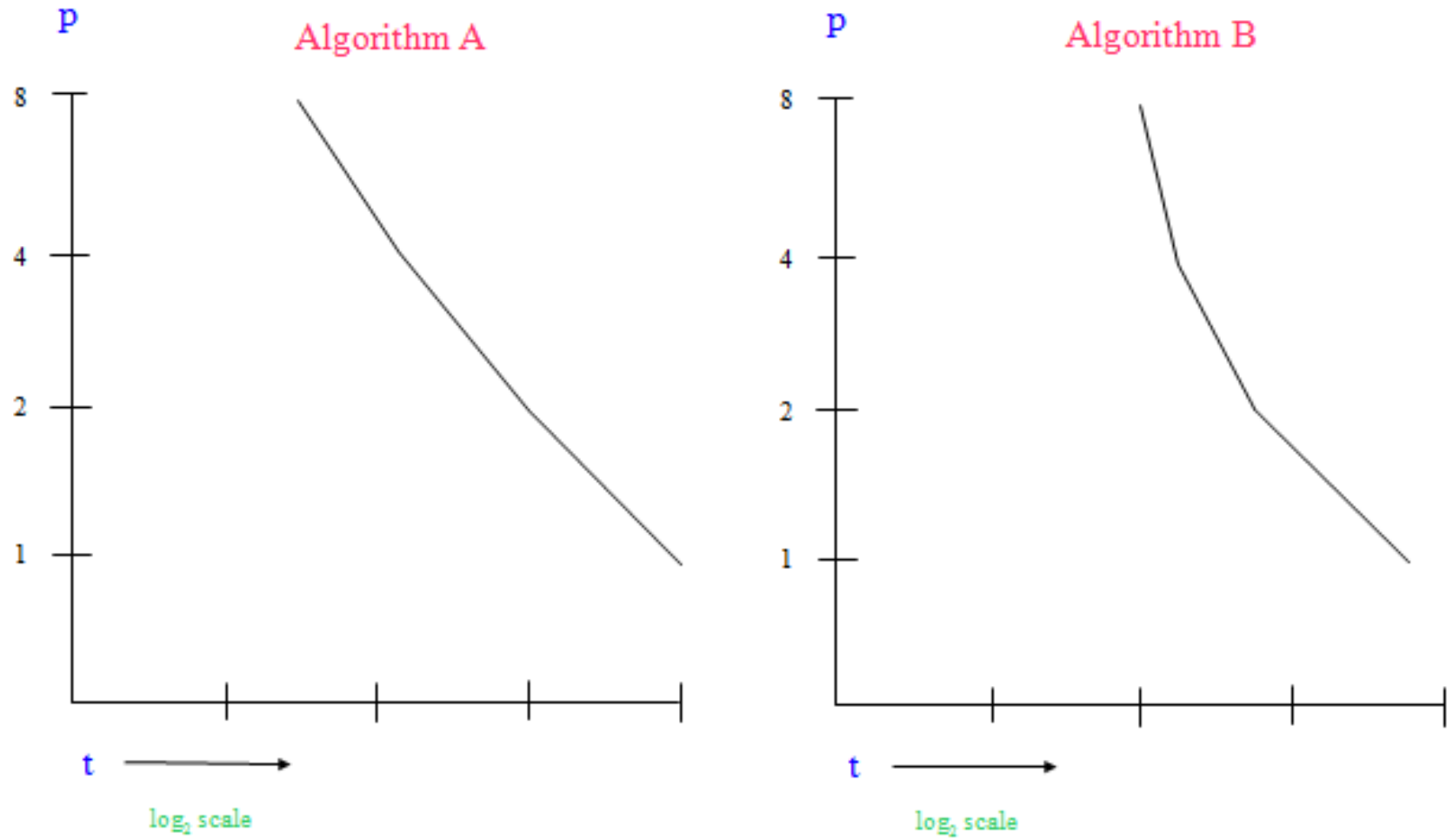


Figure 1: Which is better?

Terms and Notations

Sequential Runtime	T_1
Sequential Fraction	s
Parallel Fraction	$p = 1 - s$
Parallel Runtime	T_N
Cost	$C_N = NT_N$
Parallel Overhead	$T_o = C_N - T_1$
Speedup	$S_N = T_1/T_N$
Efficiency	$E = S_N/N$

Definitions and Assumptions

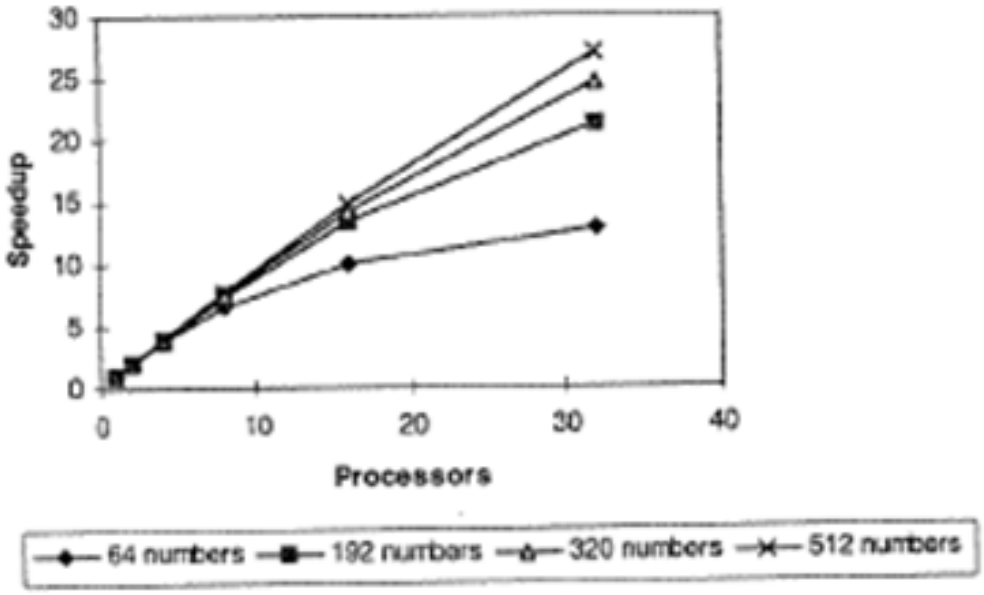
- The *sequential running time* is usually taken to be the running time of the best sequential algorithm.
- The *sequential fraction* is the part of the algorithm that is inherently sequential (reading in the data, splitting, etc.)
- The sequential part is not always a well-defined part of the computation, as we will see, so this is a conceptual notion.
- The parallel overhead includes all additional work that is done due to parallelization.
  - communication
  - nonessential work
  - idle time

Cost, Speedup, and Efficiency

- These three concepts are closely related.
- A parallel system is *cost optimal* if $C_N \in \Theta(T_1)$.
- A parallel system is said to exhibit *linear speedup* if $S \in \Theta(N)$.
- Hence, linear speedup $\Leftrightarrow$ cost optimal $\Leftrightarrow E = 1$
- If $E > 1$, this is called *super-linear speedup*.
- Superlinear speedup can arise in practice, though it it is not possible *in principle*.

Example: Parallel Semigroup Revisited

- With n data elements and N processing cores, we first combine n/N elements sequentially locally.
- Then combine local results.
- Theoretical parallel running time on a PRAM is $n/N + lgN$.
- Roughly speaking, the $\lg N$ term represents the overhead.
- The below results show an empirical measure of strong scalability of parallel semigroup for different values of n and N.



Example: Matrix Multiplication

```
1  function matmult_naive_parallel!(C, A, B)
2      Threads.@threads for i ∈ 1:size(A, 1)
3          for j ∈ 1:size(B, 2)
4              C[i, j] = A[i, :]'B[:, j]
5          end
6      end
7      return(C)
8  end
```

```
julia> Threads.nthreads()
3
julia> @benchmark matmult_naive_parallel!(C, A, B) evals=5 samples=1
  2.823 s (2000019 allocations: 15.14 GiB)
```

```
julia> Threads.nthreads()
1
julia> @benchmark matmult_naive_parallel!(C, A, B) evals=5 samples=1
  5.787 s (2000019 allocations: 15.14 GiB)
```

Factors Affecting Speedup

- Sequential Fraction (Ramp-up and Ramp-down Time)
- Parallel Overhead
  - Unecessary/duplicate work
  - Communication overhead
  - Idle time
  - Time to split/combine
- Task Granularity (how finely can the work be divided?)
- Static Task Distribution (how easy is it to divide the work evenly initially?)
- Dynamic Task Distribution (how easy is it to re-balance workload?)
- Sychronization/Data Dependency (how much do tasks depend on each other?).

Amdahl's Law

- Speedup is bounded by

$$(s+p)/(s+p/N) = 1/(s+p/N) = N/(sN+p)$$

- This means more cores $\Rightarrow$ less efficient!
- How do we combat this?
- Typically, larger problem size $\Rightarrow$ more efficient.
- This can be used to "overcome" Amdahl's Law.

Gustafson's Viewpoint

- Gustafson noted that typically the serial fraction does not increase with problem size (is this realistic?).
- This view leads to an alternative bound on speedup called scaled speedup.

$$(s + pN)/(s + p) = s + pN = N + (1 - N)s$$

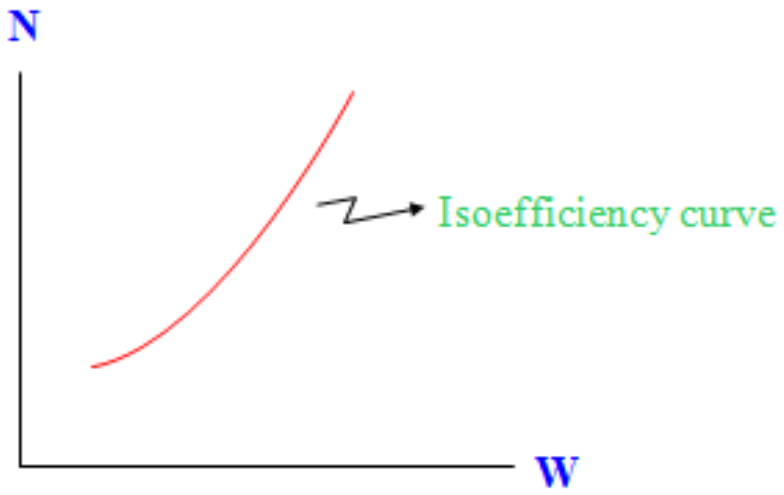
- This may be a more realistic viewpoint.

Difficulties with Measuring Strong Scalability

- Strong scalability analysis requires measuring a computation that can be completed on a single core (or a small number of cores).
- This limits the degree to which scaling can realistically be measured.
- There is a limit to the extent to which any computation that can done on a single core can be scaled.
- What we need is to scale up the computations themselves at the same time.
- This is what makes weak scalability and other related measures attractive.
- Unfortunately, these measures are even more difficult to compute.

The Isoefficiency Function

- The isoefficiency function $f(N)$ of a parallel system represents the rate at which the problem size must be increased in order to maintain a fixed efficiency



- This function is a measure of scalability that can be analyzed using asymptotic analysis.
- It may be difficult to compute for certain problems where the empirical running time is not a predictable function of size.

Weak Scalability

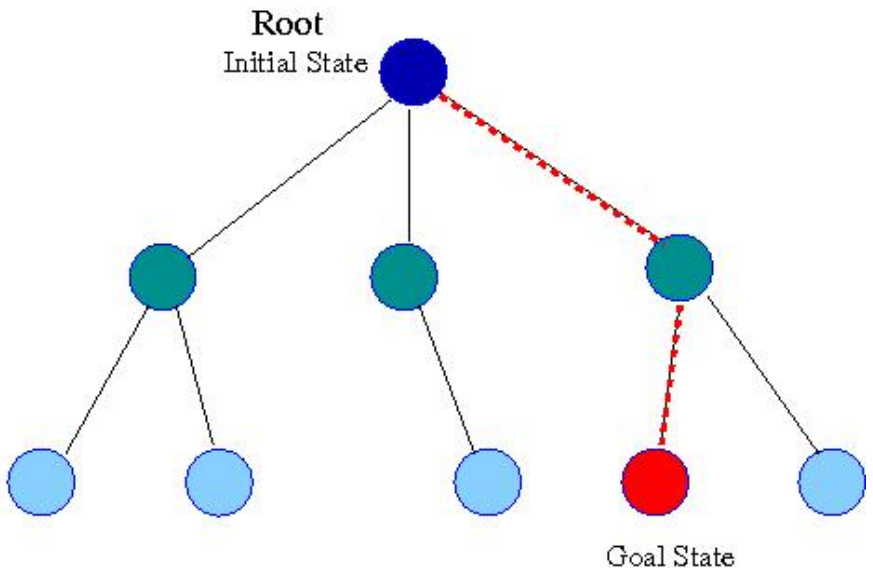
- Weak scalability is another alternative.
- Essentially, we try to see how much more computation we can do by adding cores.
- This necessitates some notion of what we mean by "more computation."
- One way is simply to scale up the size—how much bigger instances can we tackle with more cores.
- This can be done relatively easily with predictable computations like parallel semigroup.
- If we have a reliable measure of progress, another alternative is to see how much further along we can get in a given lengthy computation.
- We may also fix the total amount of core time available but vary the wallclock time in which is it delivered and see what changes.

Direct Measures of Overhead

- A final alternative is to directly measure the overhead, e.g., idle time.
- This is not quite as easy as it seems it would be, but can be done to a limited extent.
- It may be difficult to measure things like redundant work in sophisticated algorithms.

Algorithms for NP-complete Problems

- *Tree search* algorithms systematically search the nodes of a dynamically constructed acyclic graph for certain *goal nodes*.



- Tree search algorithms are used to solve NP-complete problems in many different arenas.
  - Constraint satisfaction,
  - Game search,
  - Constraint Programming, and
  - Mathematical optimization.

Tree Search Algorithm

- Tree search is not a single algorithm but an algorithmic framework.
- A generic tree search algorithm consists of the following elements:
- Elements of tree search
  - Processing method: Is this a goal node?
  - Fathoming rule: Can node can be fathomed?
  - Branching method: What are the successors of this node?
  - Search strategy: What should we work on next?
- Beginning with a root node, the algorithm consists of choosing a candidate node, processing it, and either fathoming or branching.
- During the course of the search, various information (*knowledge*) is generated and can be used to guide the search.
- This is the "guided luck" that we've talked about.
- Example: Improving the naive algorithm for SAT.

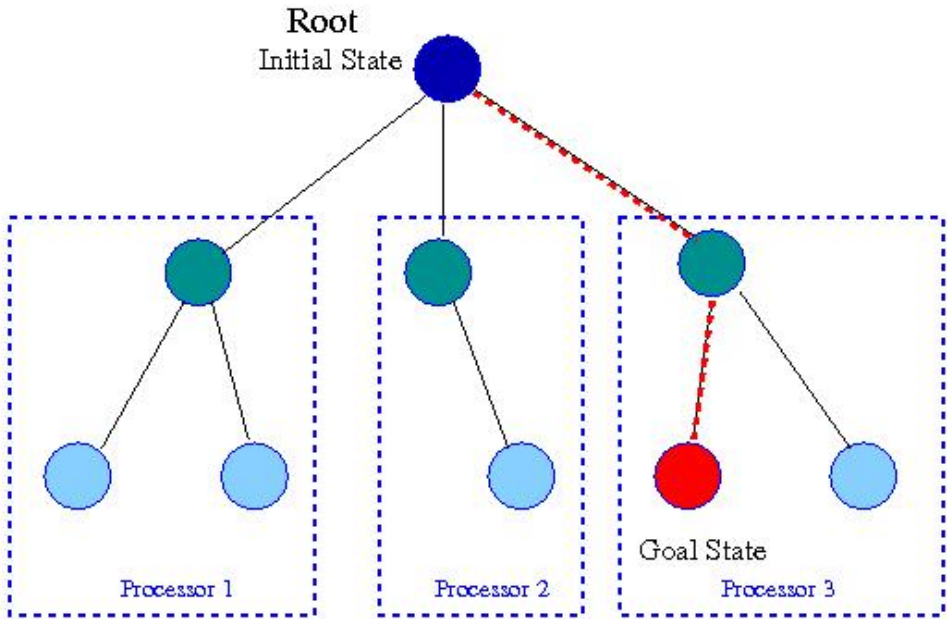
Generic Tree Search

Algorithm 1: A Generic Tree Search Algorithm

```
1  Add root node r to a priority queue Q.
2  while Q is not empty do
3      Choose a node i from Q.
4      Process the node i.
5      Apply pruning rules (can i or a successor be a goal node?)
6      if Node i can be pruned then
7          Prune (discard) node i (save i if it may be a goal node).
8      else
9          Apply successor function to node i (Branch)
10         Add the successors to Q.
```

Parallelizing Tree Search

- The generic tree search algorithm appears very easy to parallelize.
- Every time we branch, one node becomes two!
- Simply distributes nodes to the cores and continue.



- The appearance is deceiving
  - The search graph is not known a priori and could be VERY unbalanced.
  - Naïve parallelization strategies are not generally effective.
  - It's difficult to determine how to divide the available work.

Knowledge Sharing

- The goal of parallel computation is to partition a given computation into *equal parts*.
- There are two challenges implicit in achieving this goal.
  - How to partition the computation into *independent* parts.
  - How to ensure the parts are of *equal size*.
- Although partitioning is (ostensibly) easy, the parts are usually not truly independent: *knowledge-sharing* can improve efficiency.
- *Knowledge-sharing* is also necessary in order to "re-balance" when our partition turns not to consist of equal parts.
  - We need *the right data in the right place at the right time*.
  - There is a tradeoff between the *cost incurred in sharing knowledge* versus the *costs incurred by its absence*.
  - The additional cost of navigating this tradeoff is the *parallel overhead* $\Leftarrow$ **This is what we typically try to minimize**

Parallel Overhead in Tree Search

- The amount of *parallel overhead* determines the scalability.
- "Knowledge sharing" is the main driver of efficiency.
- Major components of parallel overhead in tree search
  - Communication Overhead (cost of sharing knowledge)
  - Idle Time
    - Handshaking/Synchronization (cost of sharing knowledge)
    - Task Starvation (cost of not sharing knowledge)
    - Memory Contention
    - Ramp Up Time
    - Ramp Down Time
  - Performance of Redundant Work (cost of not sharing knowledge)
- This breakdown highlights the tradeoff between centralized and decentralized knowledge storage and decision-making.

Avoiding the "Dead Ends"

- Improvements to the basic algorithm usually consist of heuristics for avoiding going too far down the "dead ends."
- This pruning of dead ends is usually done by exploiting information about other previous dead ends.
- This is why information sharing is important.
- Unfortunately, this pruning is what causes some cores to run out of work more quickly than expected.
- This is what necessitates *dynamic load balancing*.
  - Load balancing periodically re-distributes work (nodes) when cores run out.
  - This takes time and increases overhead, but is necessary to maintain efficiency.

Challenges from Tree Shape: Nice Trees

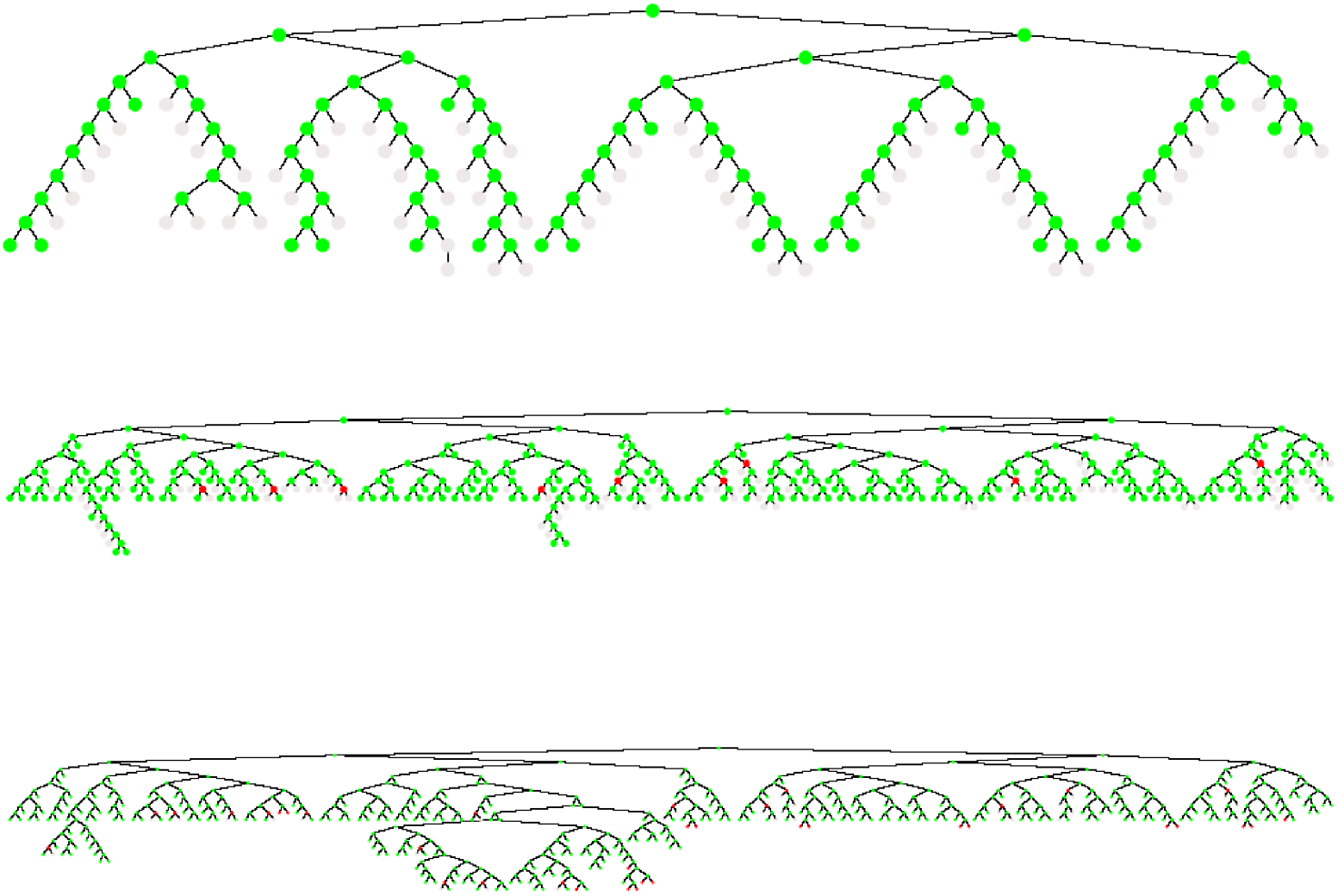


Figure 2: Branch-and-Bound Trees

Challenges from Tree Shape: Ugly Trees

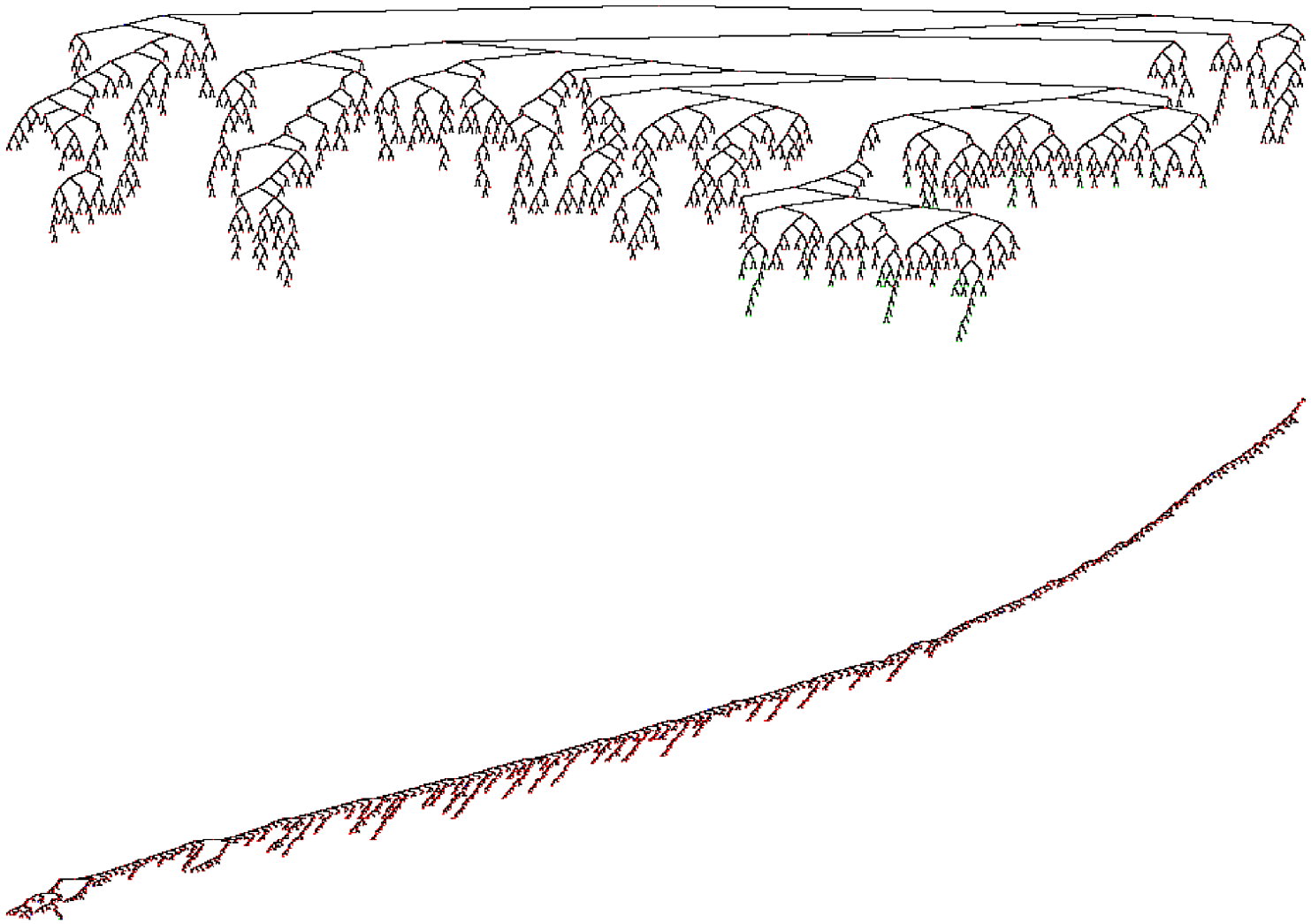


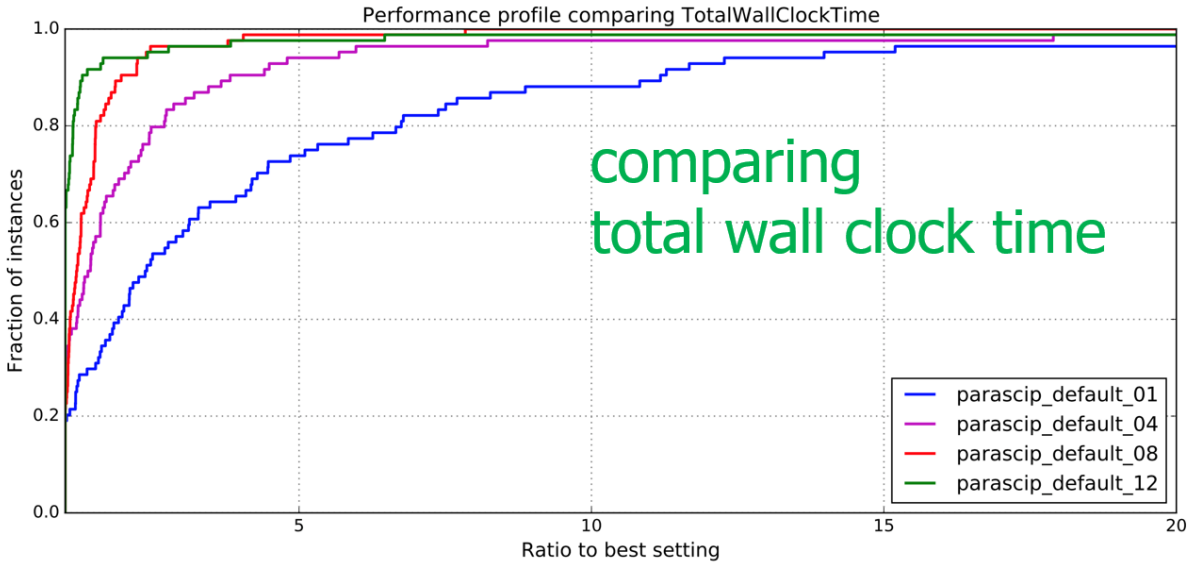
Figure 3: Branch-and-Bound Trees

Efficiency Versus Scalability

- Typically, a parallel algorithm is developed starting from an underlying sequential algorithm.
- Sequential algorithms are generally optimized for sequential efficiency.
- In parallel, scalability may be more important than efficiency.
- The sequential algorithm with the best efficiency may not be the best starting point for developing a scalable parallel algorithm.
- The optimizations leading to better sequential efficiency may actually lead to reduced scalability.

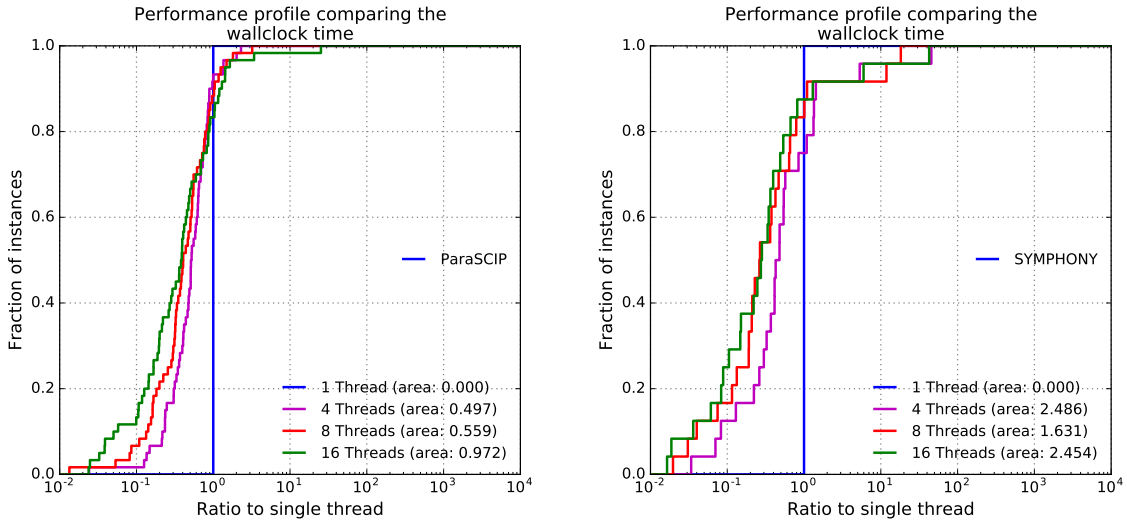
Visualizing Scalability: Performance Profiles

- Performance profiles are typically used to compare different algorithms
- They can, however, be used to compare the same algorithm under different conditions.
- For scalability, we compare with differing numbers of threads.
- A down side is that performance profiles compare to virtual best, whereas scalability compares to single-thread.



Visualizing Scalability: Scalability Profiles

- Straight performance profile considers ratios against virtual best.
- An alternative is to consider ratios against single thread.
- In the latter case, we must allow ratios less than one.



(a) ParaSCIP (b) SYMPHONY

Figure 4: Scalability profile of wallclock running time.