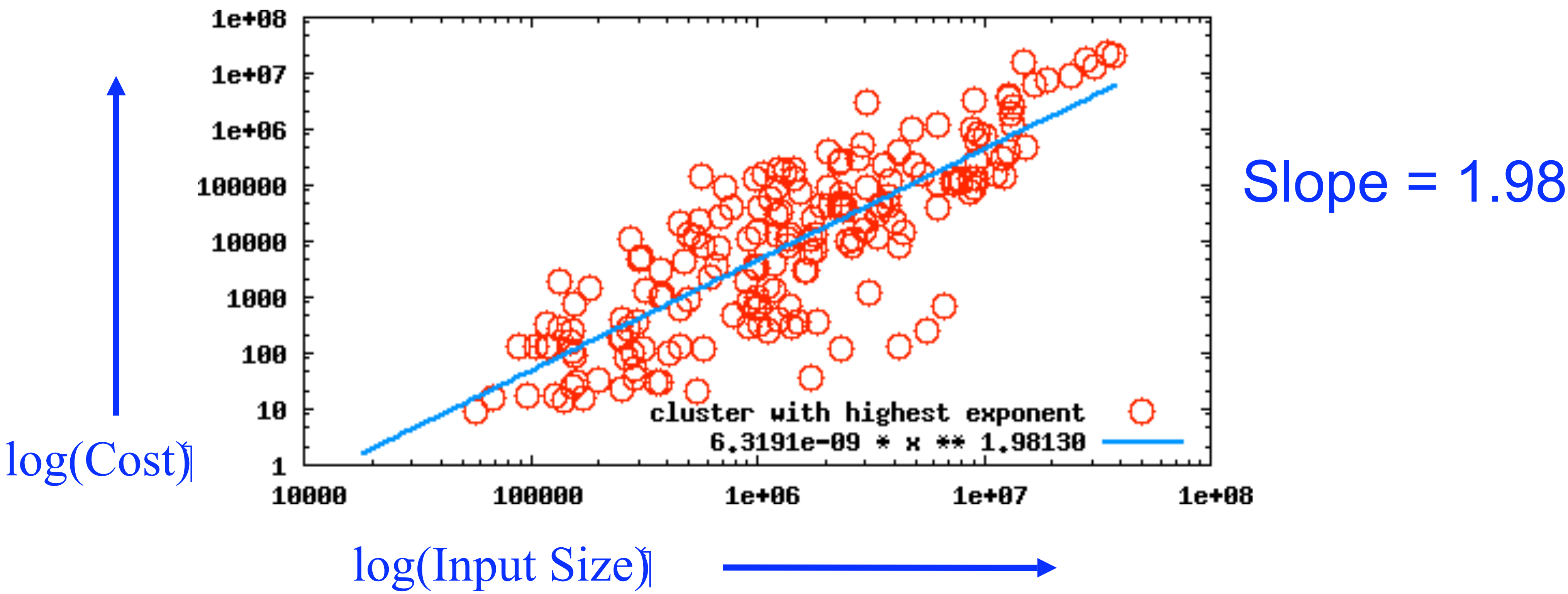
Measuring Empirical Computational Complexity Using `trend-prof`



Simon Fredrick Vicente Goldsmith
Daniel Shawcross Wilkerson
Alex Aiken

Algorithmic Scalability is a Timeless Concern

- No matter your resources, an unnecessary super-linearity can eat them all.
  - That is, never send a quadratic algorithm to do a linear algorithm's job.
- **We want** an understanding of performance that has
  - the **concreteness of empiricism** on a realistic set of workloads for a real program,
  - and the **generality of a trend** without the difficulty of theoretical analysis.

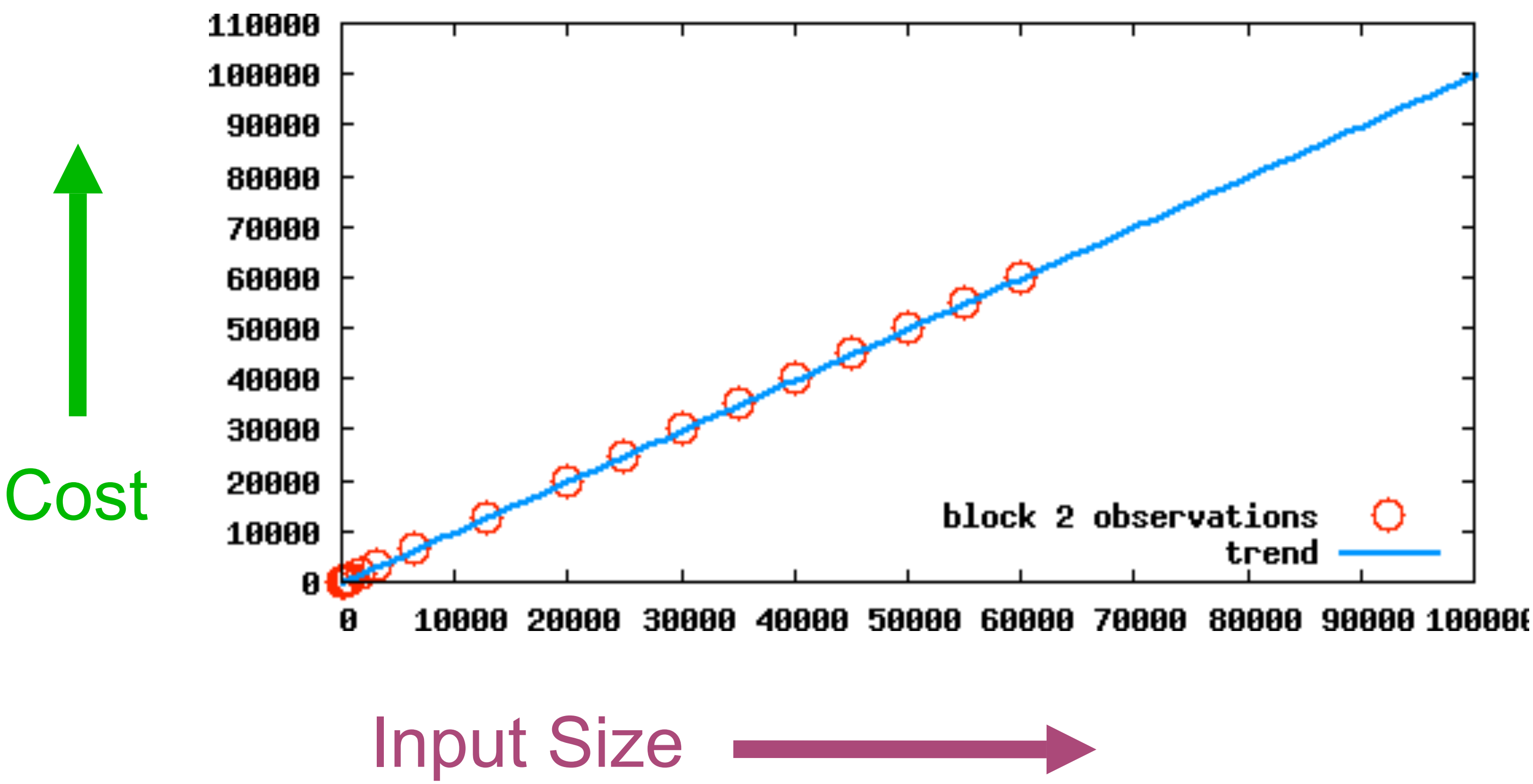
Empirical Asymptotic: Combining the Strength of Two Approaches

Consider performance of Insertion Sort

- NOT: Theoretical Asymptotic: analysis
  - worst case $\Theta(n^2)$
  - best case $\Theta(n)$
  - expected case depends on input distribution
- NOT: Empirical Pointwise: gprof
  - *e.g.*, 2% of total time
- *BUT: Empirical Asymptotic*: trend-prof
  - empirically scales as, *e.g.*, $n^{1.2}$

Core Idea

- For each line of the program
- Relate cost to input size



Our Method

- **Measure** performance, making a **matrix:**
  - one row per line of code,
  - one column per input workload.
- **Cluster** rows based on **linear correlation**.
- **Fit** rows to a **power law**.

Running Example: `bsort`

```
void bsort(int n, int *arr) {
int i=0;
while (i<n) {
    int j=i+1;
    while (j<n) {
        if (arr[i] > arr[j])
            swap(&arr[i], &arr[j]);
        j++; }
    ++i; } }
```

Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

Cost	$\texttt{Workload}_1$
$\texttt{Line}_1$	1
$\texttt{Line}_2$	61
...	
$\texttt{Line}_5$	1770
...	

Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

Cost	$\mathtt{Work(load)}_1$	$\mathtt{Work}_2$
$\mathtt{Line}_1$	1	1
$\mathtt{Line}_2$	61	201
…		
$\mathtt{Line}_5$	1770	19900
…		

Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

Cost	$\texttt{Work}_1$	$\texttt{Work}_2$	…	$\texttt{Work}_{60}$
$\texttt{Line}_1$	1	1	…	1
$\texttt{Line}_2$	61	201	…	60001
…				
$\texttt{Line}_5$	1770	19900	…	1.79997e9
…				

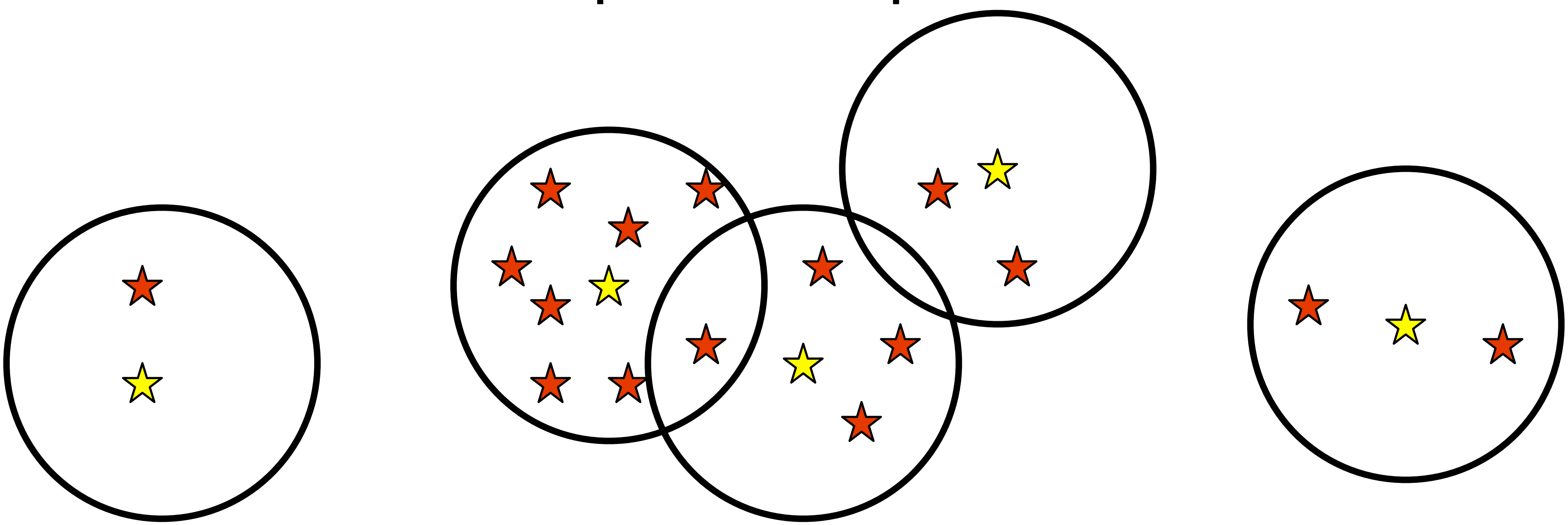
Problem 1: Too Many Lines of Code

Program	Basic Blocks
`bzip`	1032
`maximus`	1220
`elsa`	33647
`banshee`	13308

- Leads to too many results to look at
  - Observation: Many lines vary together

Solution 1: *Clusters*, Subsets of Correlated Lines of Code

- Greedily assign lines of code to all clusters whose *cluster rep* they fit with $R^2 > 0.98$
- Lines of code that don't fit any cluster rep become new cluster reps
- Initial cluster rep is the input size



Empirical Fact: Clustering Works

Program	Basic Blocks	Clusters	Costly Clusters
`bzip`	1032	23	10
`maximus`	1220	13	9
`elsa`	33647	1489	30
`banshee`	13308	859	26

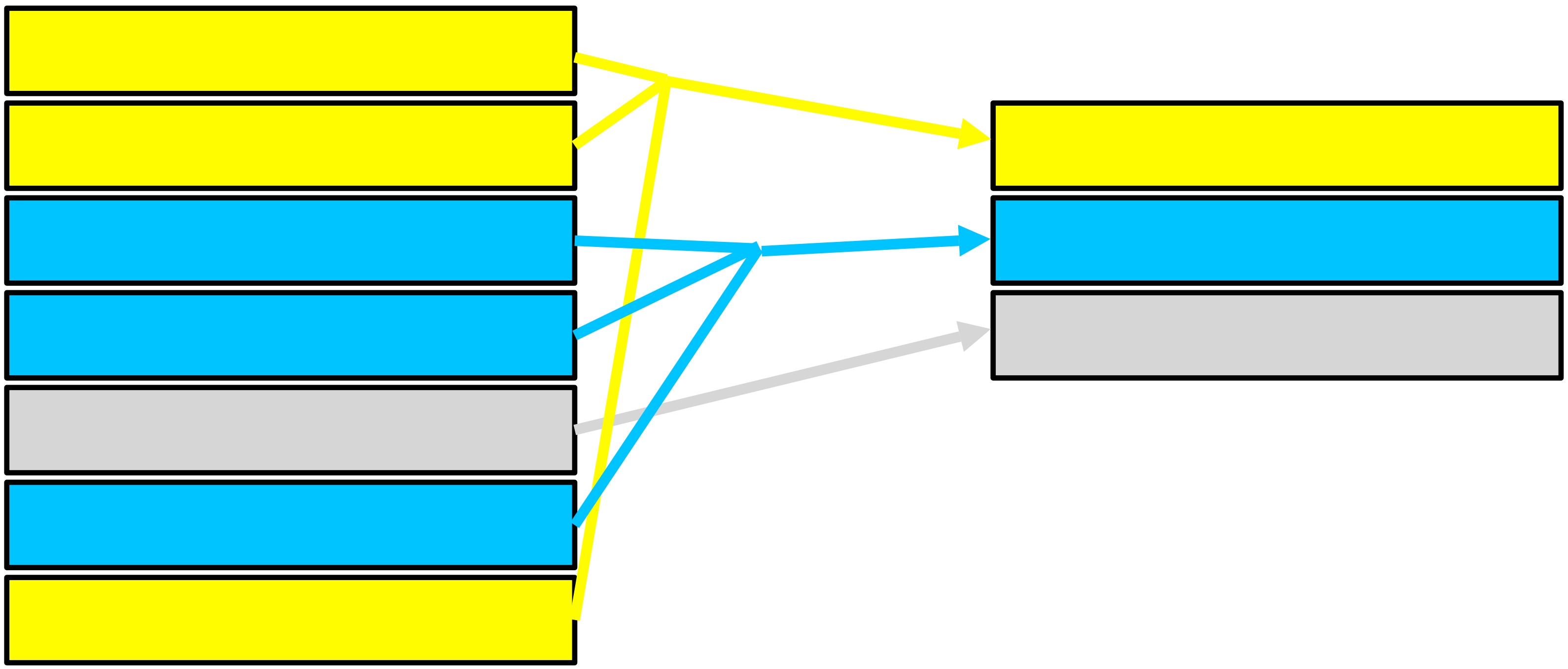
- Order of magnitude less clusters than blocks
- Furthermore there are few “costly” clusters
  - a cluster is “costly” if it accounts for more than 2% of total performance on *any* workload

Running Example: Clusters for `bsort`

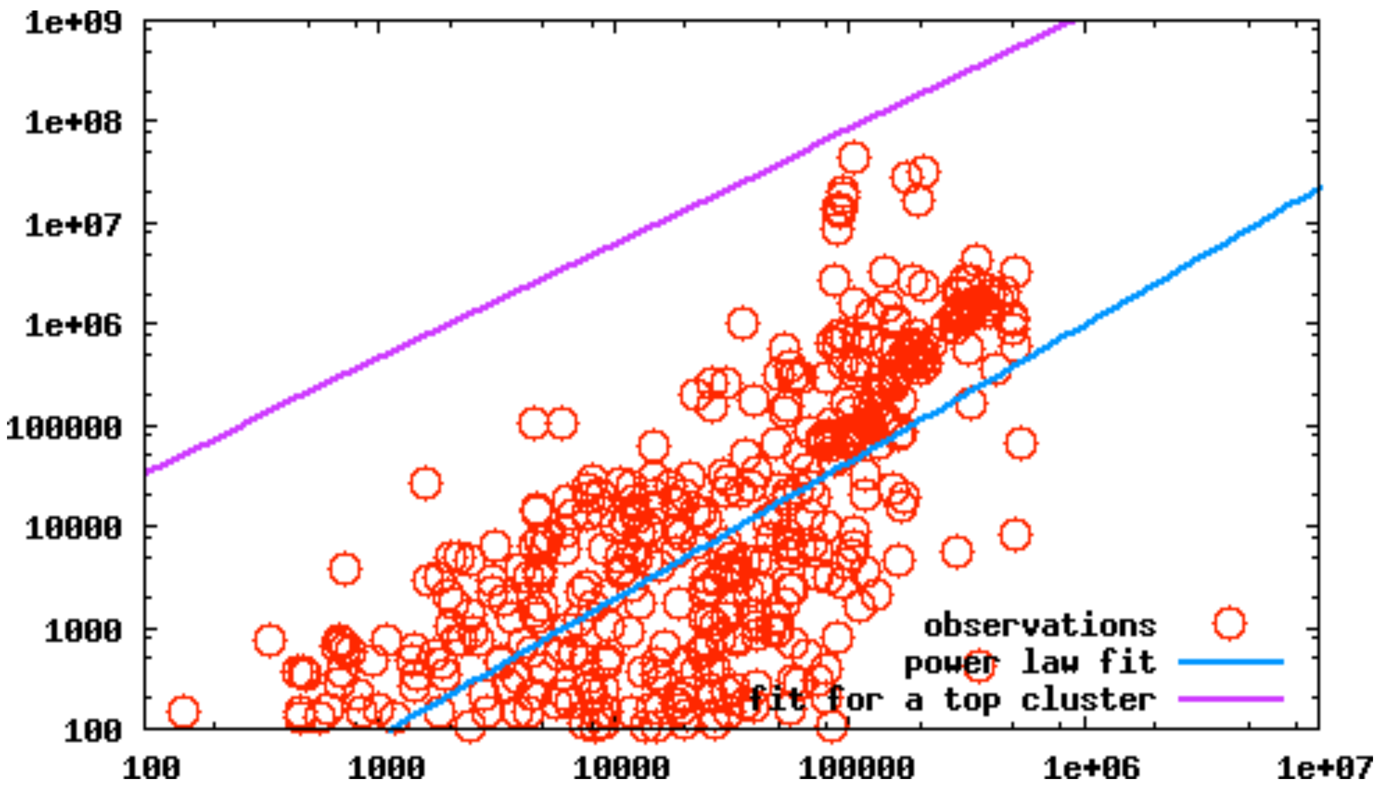
```
   void bsort(int n, int *arr) {
1: int i=0;
2: while (i<n) {
3:     int j=i+1;
4:     while (j<n) {
5:         if (arr[i] > arr[j])
6:             swap(&arr[i], &arr[j]);
7:         j++; }
8:     ++i; } }
```

From Now on, Think *Clusters* Rather Than *Lines of Code*

- Use the clusters as "abstract lines of code"
  - from now on, we just call them lines of code



Problem 2: How Do We Get a Trend From a Bunch O'Dots?



What The Heck Is This Trend ?!

Solution 2: Fit the Matrix Rows to a Power Law

Look for performance trends:

- each row records work done by each line of code

Cost	$\texttt{Work}_1$	$\texttt{Work}_2$	…	$\texttt{Work}_{60}$
$\texttt{Line}_1$	1	1	…	1
$\texttt{Line}_2$	61	201	…	60001
…				
$\texttt{Line}_5$	1770	19900	…	1.79997e9
…				

...Versus a User-Defined Notion of Input Size

Look for performance trends:

- each row records work done by each block
- with respect to user-specified input size

Cost	$\mathtt{Work}_1$	$\mathtt{Work}_2$ ...	$\mathtt{Work}_{60}$
`InputSize`	60	200 ...	60000
$\mathtt{Line}_1$	1	1 ...	1
$\mathtt{Line}_2$	61	201 ...	60001
...			
$\mathtt{Line}_5$	1770	19900 ...	1.79997e9
...			

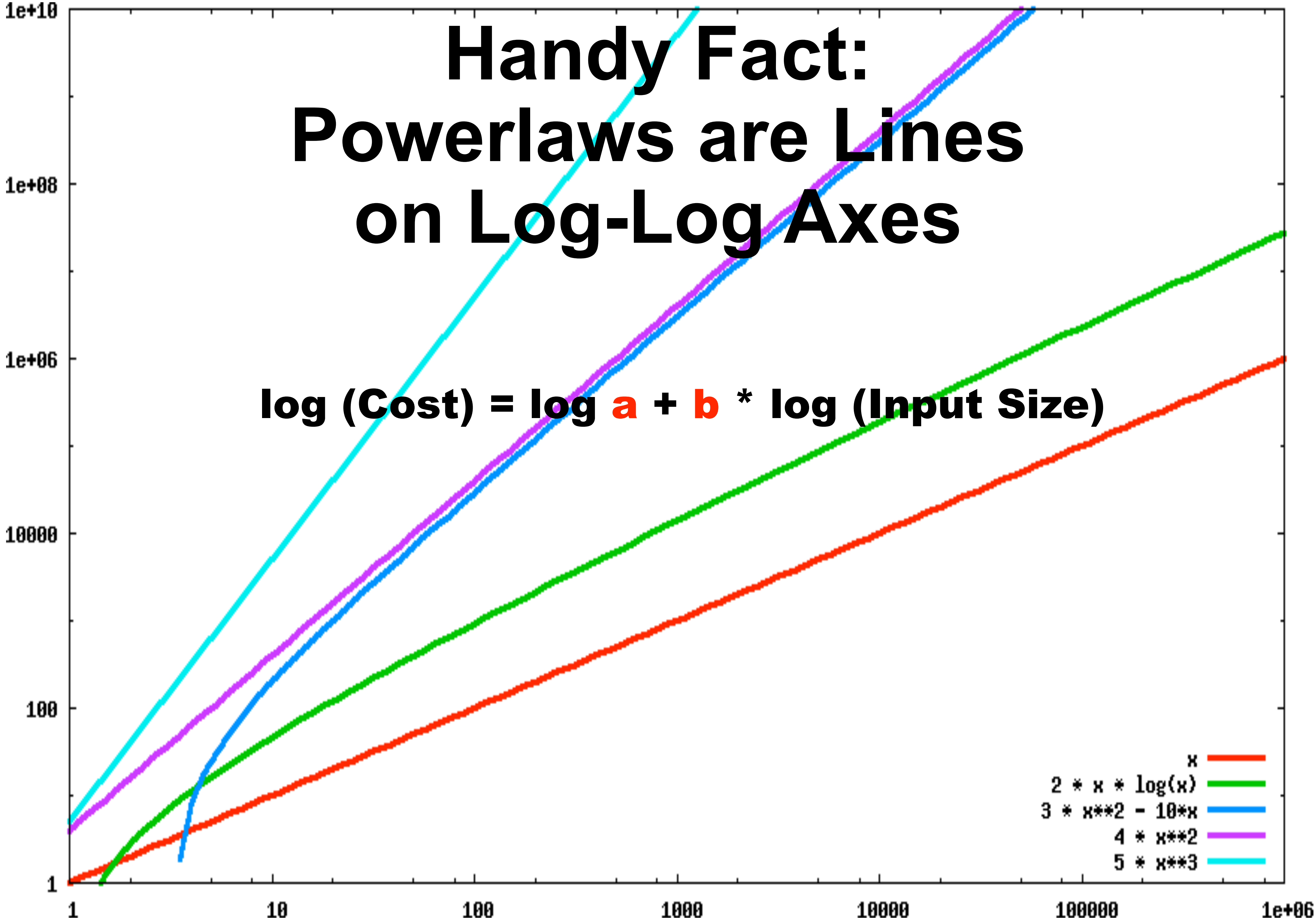
Again, Model Performance as a *Powerlaw* of Input Size

$$\textbf{(Cost)} = {\color{red}\mathbf{a}} * \textbf{(Input Size)}^{\color{red}\mathbf{b}}$$

- Low dimensional
  - gives us high confidence for less data
- Easy to interpret
- Captures the high-order term
  - logarithmic factors are quite small in practice
  - polynomials converge to high order term

Handy Fact: Powerlaws are Lines on Log-Log Axes

log (Cost) = log **a** + **b** * log (Input Size)

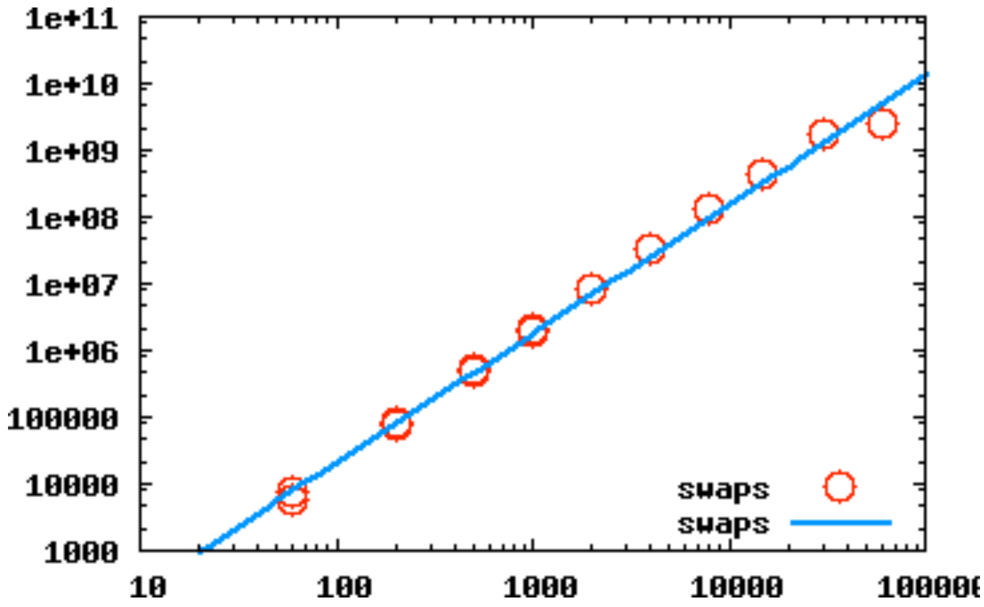


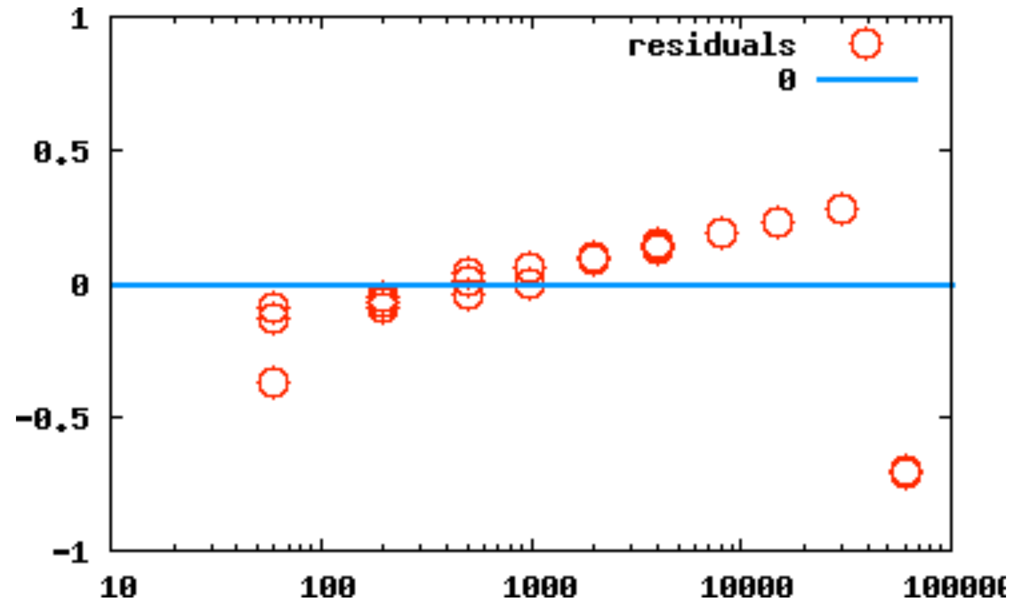
Demo

Running Example: `trend-prof` results for bsort

Max Cost in billions of basic block executions	**Cluster Name**	**Cluster Total** as a function of **Input Size**	R^2 0=bad 1=good
11	Compares	$3.1\ n^{2.00}$	1.00
2.5	Swaps	$3.0\ n^{1.93}$	0.996
< 1	Size	$22\ n^{1.00}$	1.00

bsort: Plots

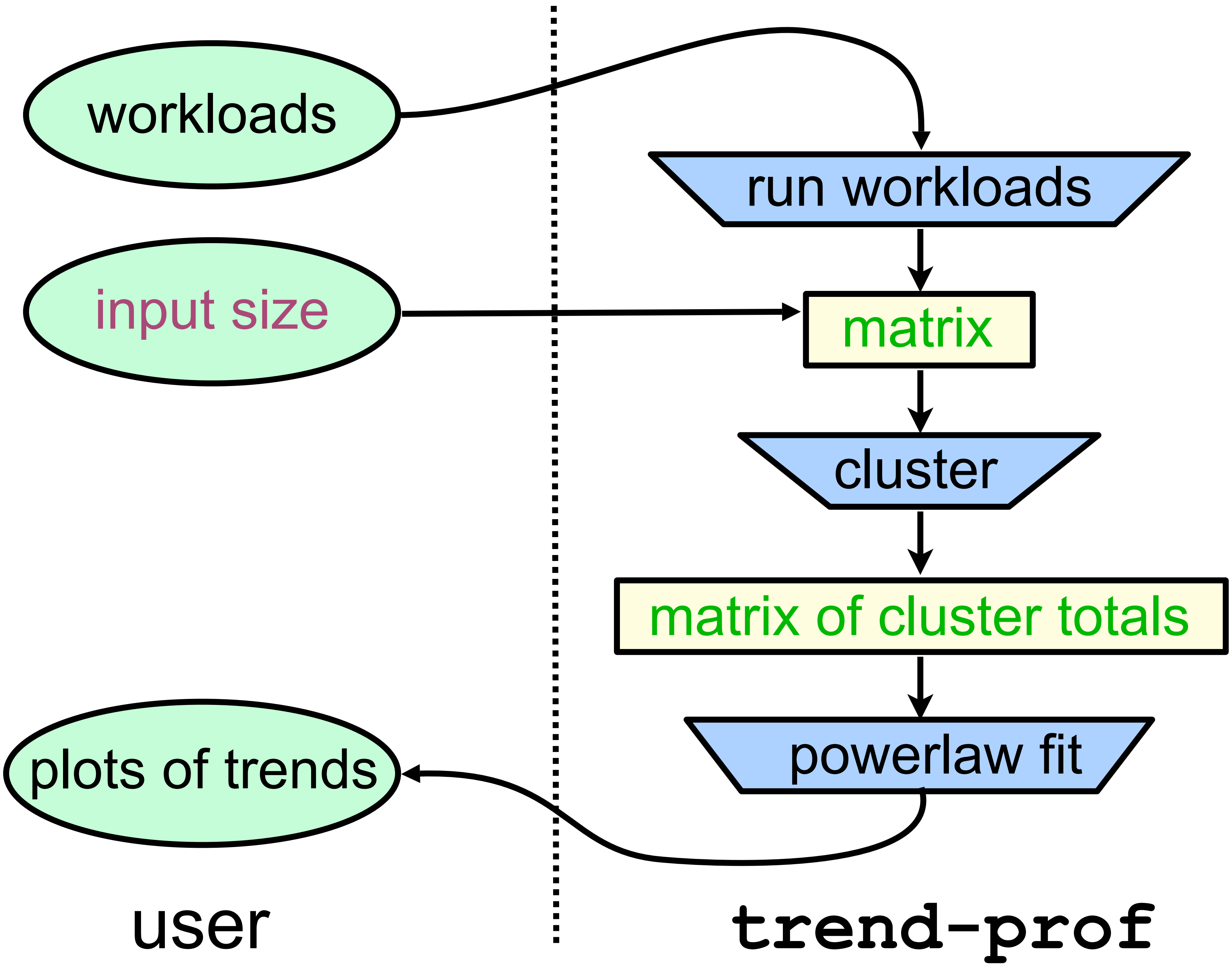




- Best-fit Plot
  - x: log(size)
  - y: log(swap cluster)
- line slope = 1.93

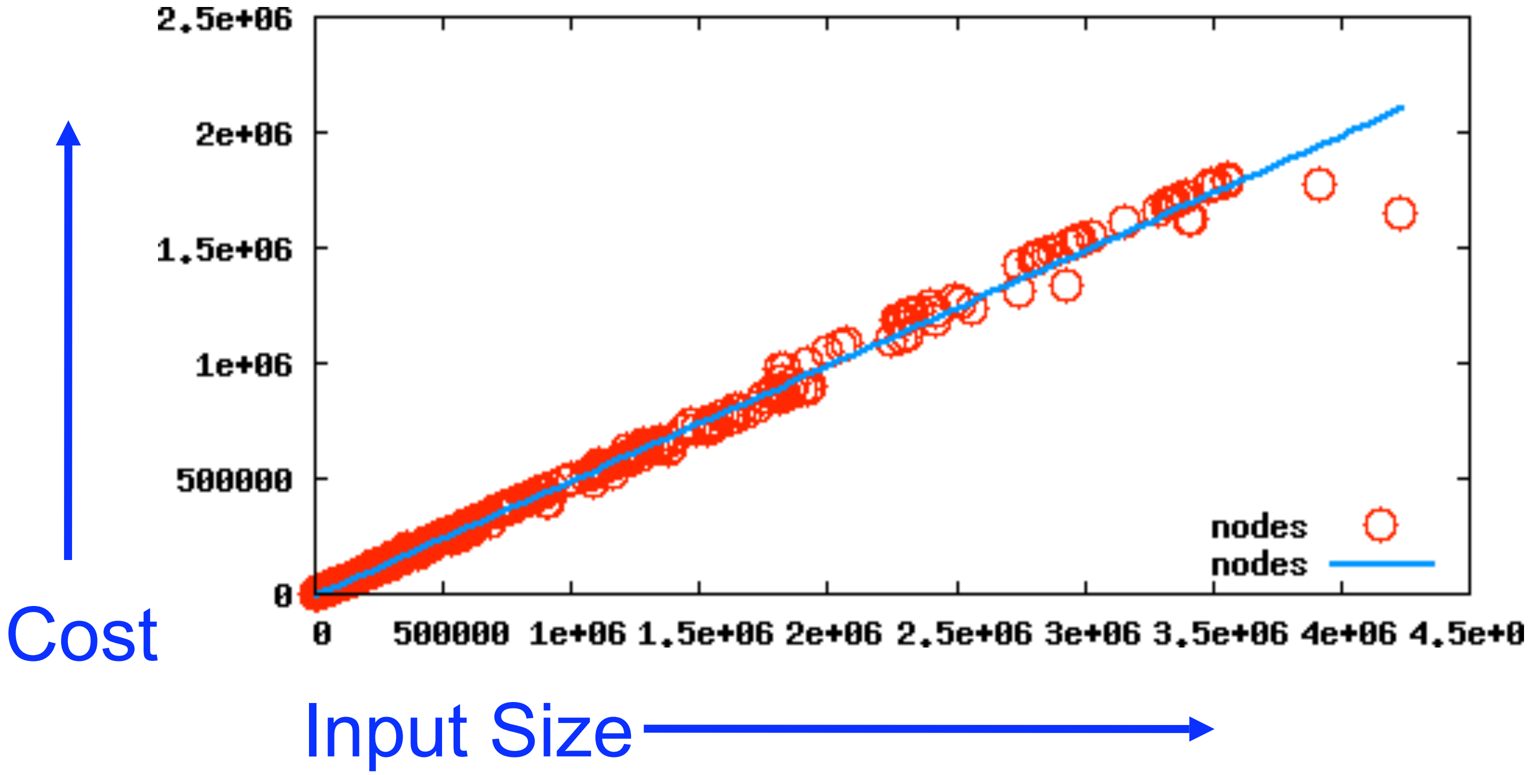
- Residuals Plot
  - y: residual
  - lack of randomness means the model missed something

`trend-prof` flow chart



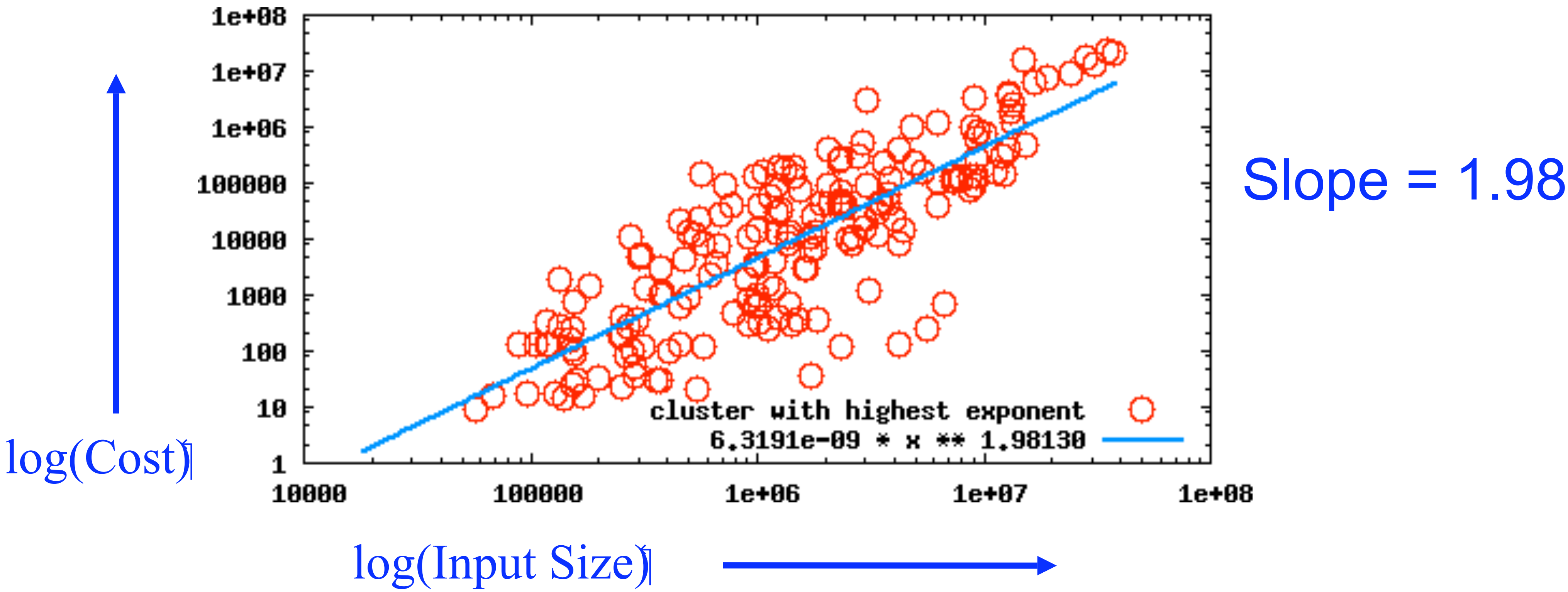
Results

Confirmed Linear Scaling



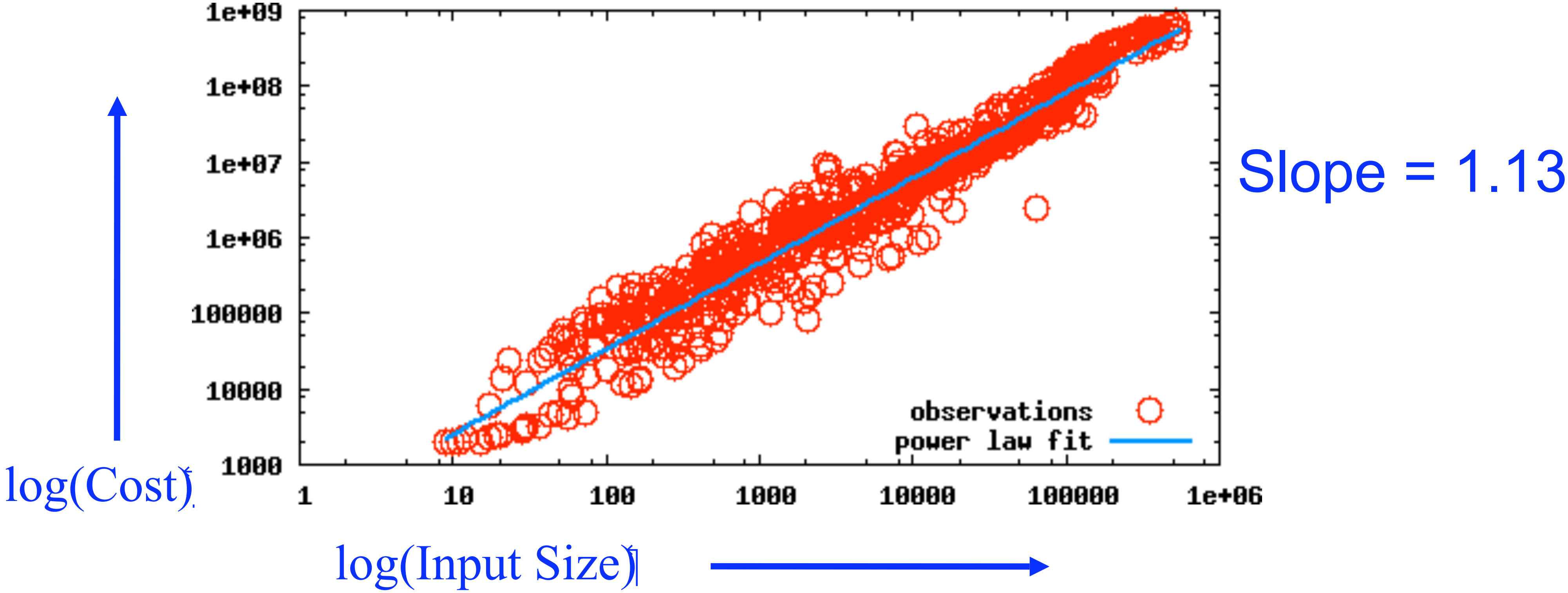
- Ukkonen's Algorithm (maximus)
  - Theoretical Complexity: $\mathcal{O}(n)$
  - Empirical Complexity: ~ n

Empirical Complexity: Andersen's



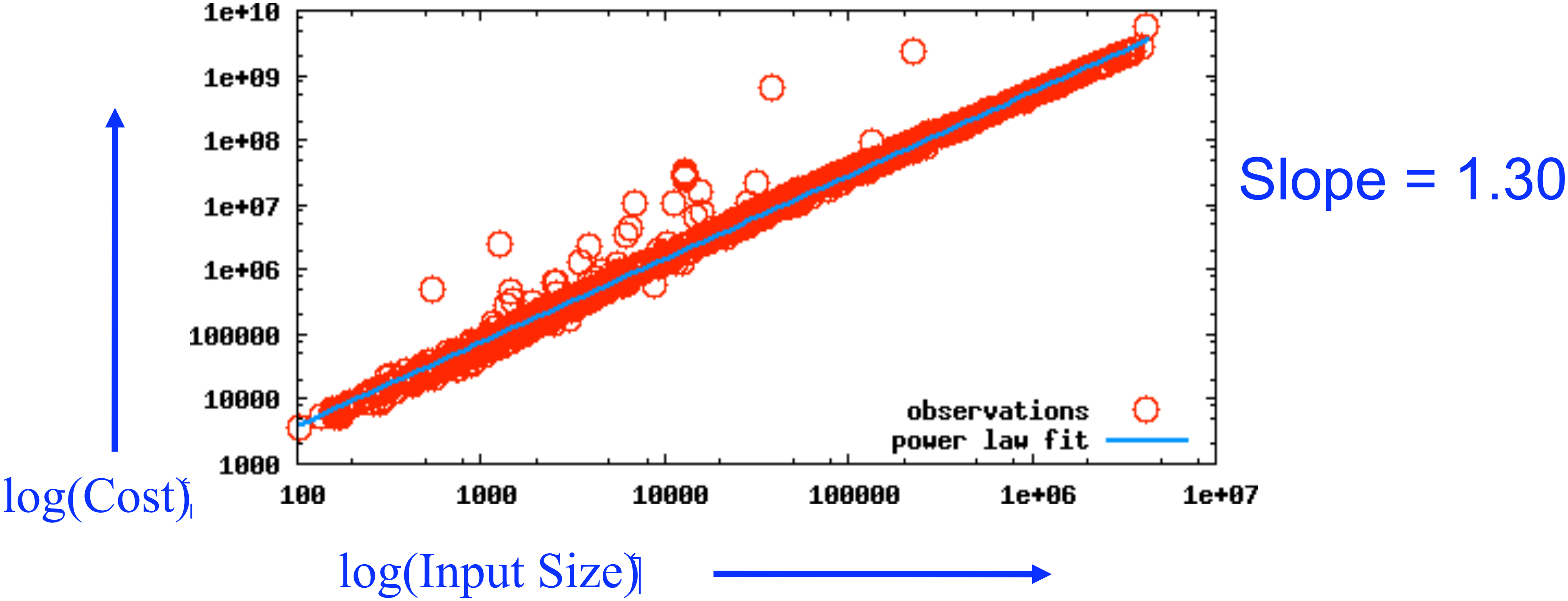
- Andersen's points-to analysis (banshee)
  - Theoretical Complexity: $\mathcal{O}(n^3)$
  - Empirical Complexity: $\sim n^{1.98}$

Empirical Complexity: GLR



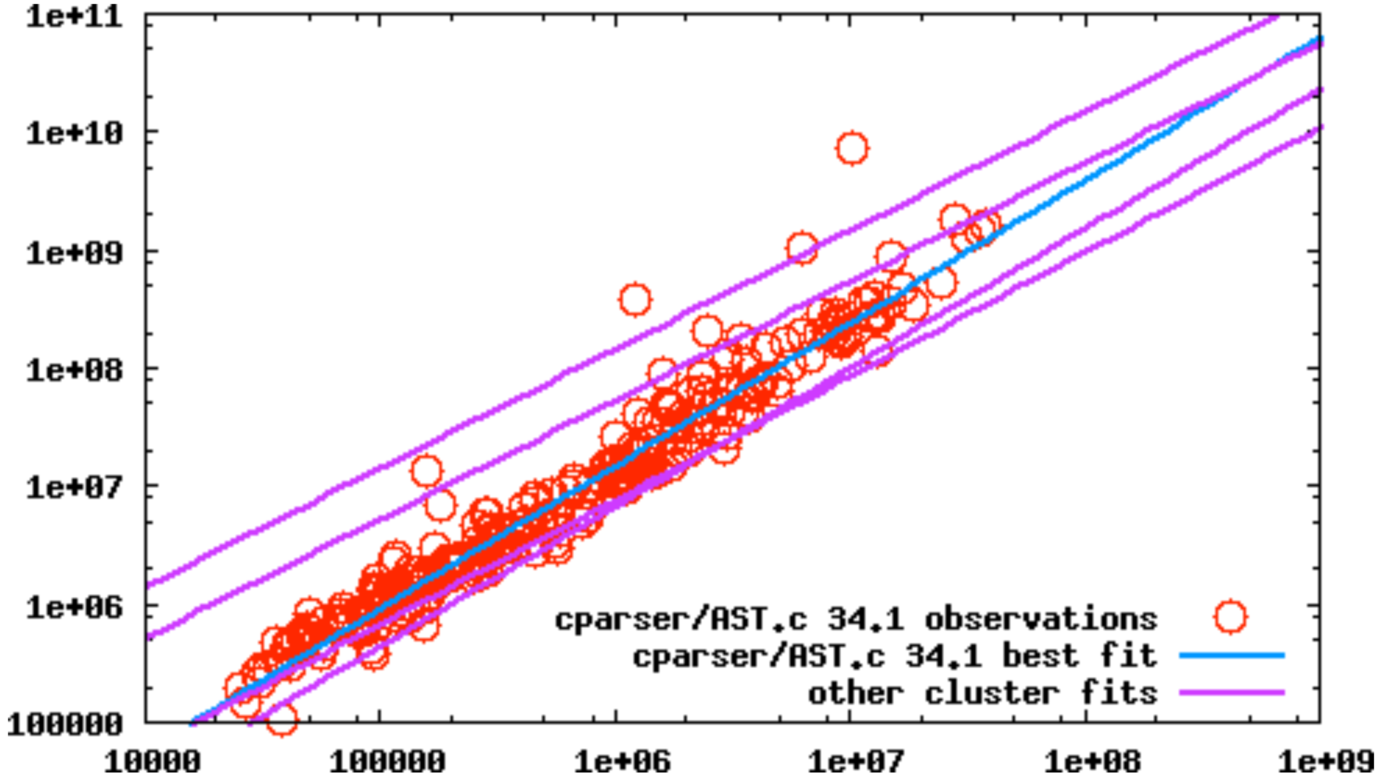
- GLR C++ parser (elkhound / elsa)
  - Theoretical Complexity: $\mathcal{O}(n^3)$
  - Empirical Complexity: $\sim n^{1.13}$

How well do you know your code?



- Output routines (maximus)
  - Theoretical Complexity: $\mathcal{O}(n)$?
  - Empirical Complexity: $\sim n^{1.30}$

Algorithms in context

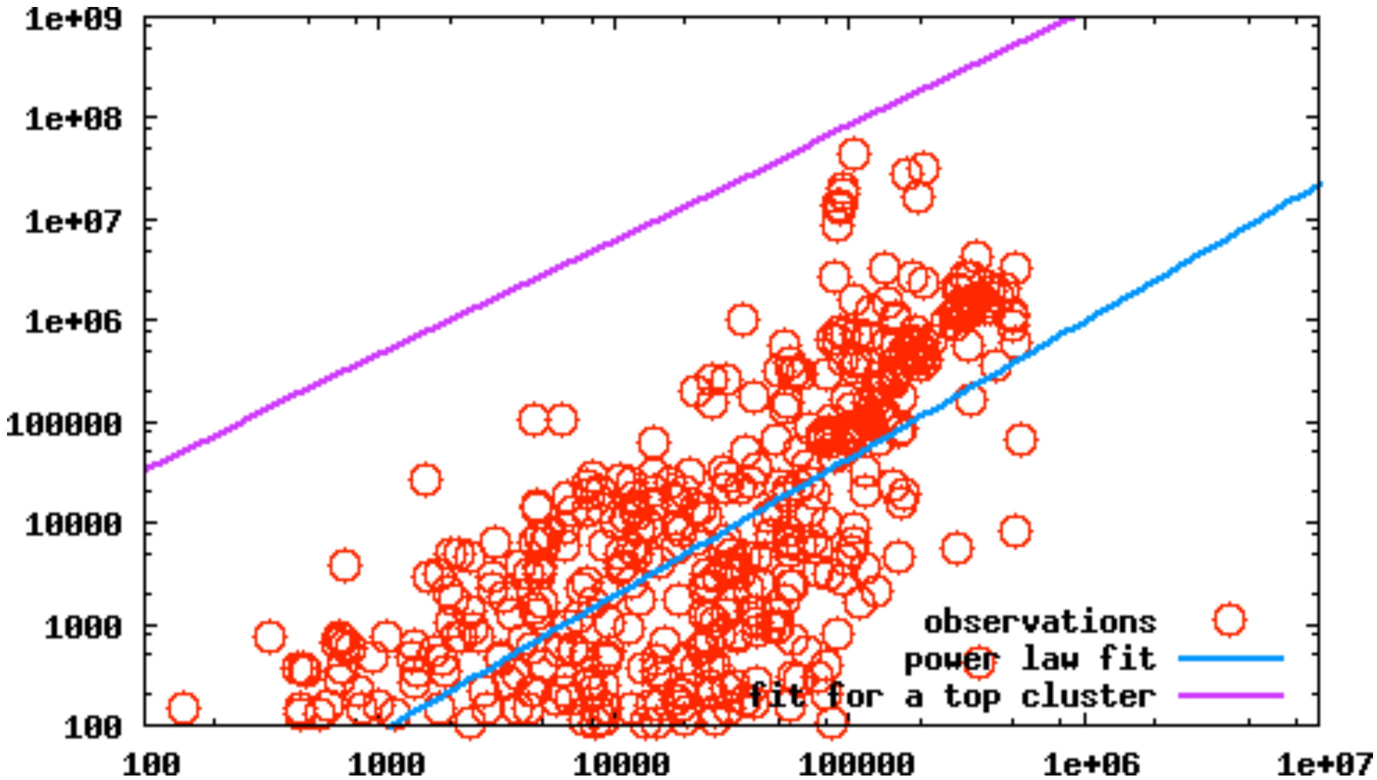


Slope = 1.21

$R^2 = 0.95$

- The linear-time list append in banshee's parser is a bug

Algorithms in Context



$R^2 = 0.65$

- The linear-time list append in Elsa's name lookup code is not a bug

Results Recap

- Confirmed linear scaling (maximus)
- Empirical scalability (Andersen's, GLR)
- Unexpected behavior (maximus)
- Algorithms in context (elsa, banshee)
  - found a performance bug in banshee's parser
  - found similar situation, but no bug in elsa

Technical Contributions of Part 1

- Built **`trend-prof`**
  - a tool to measure empirical computational complexity
- Discovered the following empirical facts
  - programs have few clusters, fewer costly ones
  - powerlaw fits work well
- Showed that powerlaw fits of basic block counts reveal general trends
  - low-dimension but still nice precision
  - plots reveal the subtleties of actual computation...

Conclusion

- Semi-automatically empirically modeling performance trends as a function of input size works
  - examining the matrix rows instead of columns yields insight into scalability
  - control flow suggests precise models that sometimes improve upon direct models
- Comparing these models to expectation finds bugs or finds properties of the data
- Trend-prof belongs in the toolbox for performance / scalability testing

Thanks

- To Alex Aiken for all the stuff advisors do
- To Daniel Wilkerson for wonderful suggestions for improving this and other talks as well as our collaboration

Questions?

Code

`trend-prof.tigris.org`

Publications

S. F. Goldsmith, A. S. Aiken, D. S. Wilkerson. Measuring Empirical Computational Complexity. FSE 2007.

S. F. Goldsmith. Measuring Emprical Computational Complexity. PhD dissertation. UC Berkeley. 2009.