Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms

Jason Ansel Yee Lok Wong Cy Chan Marek Olszewski
Alan Edelman Saman Amarasinghe

MIT - CSAIL

April 4, 2011

Outline

A motivating example

- How would you write a *fast* sorting algorithm?

A motivating example

- How would you write a *fast* sorting algorithm?
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort

A motivating example

- How would you write a *fast* sorting algorithm?
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort
  - Binary tree sort, Bitonic sort, Bubble sort, Bucket sort, Burstsort, Cocktail sort, Comb sort, Counting Sort, Distribution sort, Flashsort, Heapsort, Introsort, Library sort, Odd-even sort, Postman sort, Samplesort, Selection sort, Shell sort, Stooge sort, Strand sort, Timsort?

A motivating example

- How would you write a *fast* sorting algorithm?
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort
  - Binary tree sort, Bitonic sort, Bubble sort, Bucket sort, Burstsort, Cocktail sort, Comb sort, Counting Sort, Distribution sort, Flashsort, Heapsort, Introsort, Library sort, Odd-even sort, Postman sort, Samplesort, Selection sort, Shell sort, Stooge sort, Strand sort, Timsort?
- Poly-algorithms

std::stable_sort

/usr/include/c++/4.5.2/bits/stl_algo.h lines 3350-3367

```
/// This is a helper function for the stable sorting routines.
template<typename _RandomAccessIterator>
  void
  __inplace_stable_sort(_RandomAccessIterator __first,
                        _RandomAccessIterator __last)
  {
    if (__last - __first < 15)
      {
        std::__insertion_sort(__first, __last);
        return;
      }
    _RandomAccessIterator __middle = __first + (__last - __first) / 2;
    std::__inplace_stable_sort(__first, __middle);
    std::__inplace_stable_sort(__middle, __last);
    std::__merge_without_buffer(__first, __middle, __last,
                                __middle - __first,
                                __last - __middle);
  }
```

std::stable_sort

/usr/include/c++/4.5.2/bits/stl_algo.h lines 3350-3367

```
/// This is a helper function for the stable sorting routines.
template<typename _RandomAccessIterator>
  void
  __inplace_stable_sort(_RandomAccessIterator __first,
                        _RandomAccessIterator __last)
  {
    if (__last - __first < 15)
      {
        std::__insertion_sort(__first, __last);
        return;
      }
    _RandomAccessIterator __middle = __first + (__last - __first) / 2;
    std::__inplace_stable_sort(__first, __middle);
    std::__inplace_stable_sort(__middle, __last);
    std::__merge_without_buffer(__first, __middle, __last,
                                __middle - __first,
                                __last - __middle);
  }
```

std::sort

/usr/include/c++/4.5.2/bits/stl_algo.h lines 2163-2167

```
/**
 *  @doctodo
 *  This controls some aspect of the sort routines.
*/
enum { _S_threshold = 16 };
```

std::sort

/usr/include/c++/4.5.2/bits/stl_algo.h lines 2163-2167

```
/**
 *  @doctodo
 *  This controls some aspect of the sort routines.
*/
enum { _S_threshold = 16 };
```

- Why 16? Why 15?

std::sort

/usr/include/c++/4.5.2/bits/stl_algo.h lines 2163-2167

```
/**
 *  @doctodo
 *  This controls some aspect of the sort routines.
*/
enum { _S_threshold = 16 };
```

- Why 16? Why 15?
- Dates back to at least 2000 (Jun 2000 SGI release)
- Still in current C++ STL shipped with GCC
- 10+ years of of `_S_threshold = 16`

Is 15 the right number?

- The best cutoff (CO) changes
- Depends on competing costs:
  - Cost of computation ($<$ operator, call overhead, etc)
  - Cost of communication (swaps)
  - Cache behavior (misses, prefetcher, locality)

Is 15 the right number?

- The best cutoff (CO) changes
- Depends on competing costs:
  - Cost of computation ($<$ operator, call overhead, etc)
  - Cost of communication (swaps)
  - Cache behavior (misses, prefetcher, locality)

- Sorting 100000 doubles with `std::stable_sort`:
  - $CO \approx 200$ optimal on a Phenom 905e (15% speedup over $CO = 15$)
  - $CO \approx 400$ optimal on a Opteron 6168 (15% speedup over $CO = 15$)
  - $CO \approx 500$ optimal on a Xeon E5320 (34% speedup over $CO = 15$)
  - $CO \approx 700$ optimal on a Xeon X5460 (25% speedup over $CO = 15$)
- Compiler's hands are tied, it is stuck with 15

Back to our motivating example

- How would you write a *fast* sorting algorithm?
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort
  - Binary tree sort, Bitonic sort, Bubble sort, Bucket sort, Burstsort, Cocktail sort, Comb sort, Counting Sort, Distribution sort, Flashsort, Heapsort, Introsort, Library sort, Odd-even sort, Postman sort, Samplesort, Selection sort, Shell sort, Stooge sort, Strand sort, Timsort?
- Poly-algorithms

Back to our motivating example

- How would you write a *fast* sorting algorithm?
  - Insertion sort
  - Quick sort
  - Merge sort
  - Radix sort
  - Binary tree sort, Bitonic sort, Bubble sort, Bucket sort, Burstsort, Cocktail sort, Comb sort, Counting Sort, Distribution sort, Flashsort, Heapsort, Introsort, Library sort, Odd-even sort, Postman sort, Samplesort, Selection sort, Shell sort, Stooge sort, Strand sort, Timsort?
- Poly-algorithms

Answer

It depends!

Autotuned parallel sorting algorithms

- On a Xeon E7340 (2×4 cores)
  1. Insertion sort `below 600`
  2. Quick sort `below 1420`
  3. 2-way parallel merge sort

Autotuned parallel sorting algorithms

- On a Xeon E7340 (2×4 cores)
  1. Insertion sort `below 600`
  2. Quick sort `below 1420`
  3. 2-way parallel merge sort
- On a Sun Fire T200 Niagara (8 cores)
  1. 16-way merge sort *below 75*
  2. 8-way merge sort *below 1461*
  3. 4-way merge sort *below 2400*
  4. 2-way parallel merge sort

Autotuned parallel sorting algorithms

- On a Xeon E7340 (2×4 cores)
  1. Insertion sort `below 600`
  2. Quick sort `below 1420`
  3. 2-way parallel merge sort
- On a Sun Fire T200 Niagara (8 cores)
  1. 16-way merge sort *below 75*
  2. 8-way merge sort *below 1461*
  3. 4-way merge sort *below 2400*
  4. 2-way parallel merge sort
- 235% slowdown running Niagara algorithm on the Xeon
- 8% slowdown running Xeon algorithm on the Niagara

Autotuned parallel sorting algorithms

- On a Xeon E7340 (2×4 cores)
  1. Insertion sort `below 600`
  2. Quick sort `below 1420`
  3. 2-way parallel merge sort
- On a Sun Fire T200 Niagara (8 cores)
  1. 16-way merge sort *below 75*
  2. 8-way merge sort *below 1461*
  3. 4-way merge sort *below 2400*
  4. 2-way parallel merge sort
- 235% slowdown running Niagara algorithm on the Xeon
- 8% slowdown running Xeon algorithm on the Niagara
- Need a way to express these algorithmic choices to enable autotuning

Outline

Algorithmic choices

Language

```
either {
  InsertionSort(out, in);
} or {
  QuickSort(out, in);
} or {
  MergeSort(out, in);
} or {
  RadixSort(out, in);
}
```

Algorithmic choices

Language

```
either {
  InsertionSort(out, in);
} or {
  QuickSort(out, in);
} or {
  MergeSort(out, in);
} or {
  RadixSort(out, in);
}
```

⇒

Representation

Decision tree synthesized by our evolutionary algorithm (EA)

The PetaBricks language

- Choices expressed in the language
  - High level algorithmic choices
  - Dependency-based synthesized outer control flow
  - Parallelization strategy
- Programs automatically adapt to their environment
  - Tuned using our bottom-up evaluation algorithm
  - Offline autotuner or always-on online autotuner

Outline

Variable accuracy algorithms

- Many problems don't have a single correct answer

Variable accuracy algorithms

- Many problems don't have a single correct answer
- Soft computing
  - Approximation algorithms for NP-hard problems
- DSP algorithms
  - Different grid resolutions
  - Data precisions
- Iterative algorithms
  - Choosing convergence criteria

Variable accuracy example

Example

```
...
for(int i = 0; i < 100; ++i) {
  SORIteration(tmp);
}
...
```

Variable accuracy example

Example

```
...
for(int i = 0; i < 100; ++i) {
  SORIteration(tmp);
}
...
```

- Competing objectives of *performance* and *accuracy*
- Must maximize performance while meeting accuracy targets

Accuracy metrics and for_enough loops

Language

```
accuracy_metric MyRMSError
```

Accuracy metrics and for_enough loops

Language

```
accuracy_metric MyRMSError

...
for_enough {
  SORIteration(tmp);
}
```

Accuracy metrics and for_enough loops

Language

```
accuracy_metric MyRMSError
...
for_enough {
  SORIteration(tmp);
}
```

$\Rightarrow$

Representation

Function from problem size to number of iterations synthesized by our EA

Accuracy variables

Language

```
accuracy_metric MyRMSError
accuracy_variable k

...
for(int i=0; i<k; ++i) {
  SORIteration(tmp);
}
```

Accuracy variables

Language

```
accuracy_metric MyRMSError
accuracy_variable k

...
for(int i=0; i<k; ++i) {
  SORIteration(tmp);
}
```

⇒

Representation

Function from problem size to k synthesized by our EA

Variable accuracy and algorithmic choices

Language

```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```

Outline

Traditional evolution algorithm

Initial population	?	?	?	?

Cost = 0

Traditional evolution algorithm

Initial population	72.7s	?	?	?

Cost = 72.7

Traditional evolution algorithm

Initial population	72.7s	10.5s	?	?

Cost = 83.2

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	?

Cost = 87.3

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s

Cost = 118.5

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost = 3.2

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost = 3.2

- Cost of autotuning front-loaded in initial (unfit) population
- We could speed up tuning if we start with a faster initial population

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost = 3.2

- Cost of autotuning front-loaded in initial (unfit) population
- We could speed up tuning if we start with a faster initial population

Key insight

Smaller input sizes can be used to form better initial population

Bottom-up evolutionary algorithm

- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

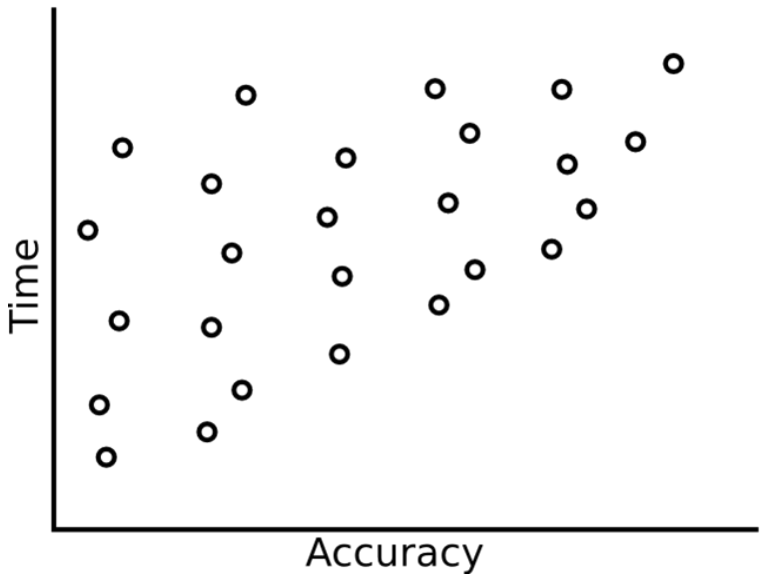
- Train on input size 1, to form initial population for:
- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

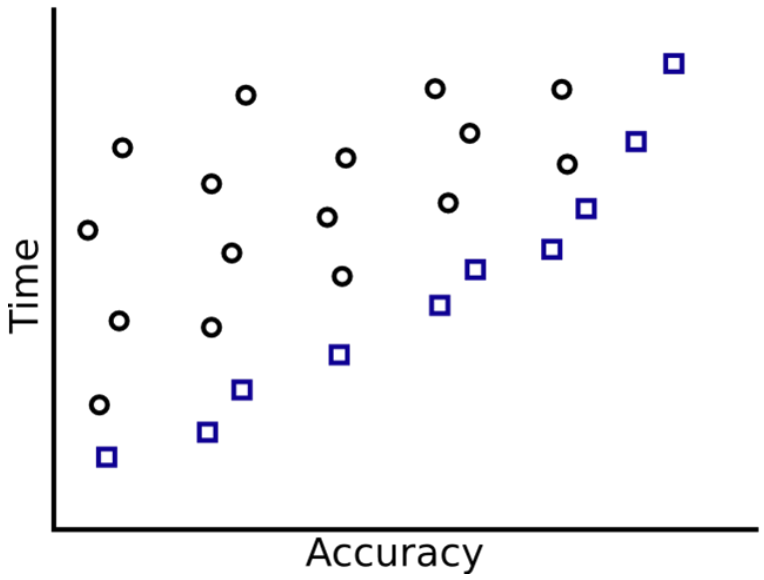
- Train on input size 1, to form initial population for:
- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

- Naturally exploits optimal substructure of problems

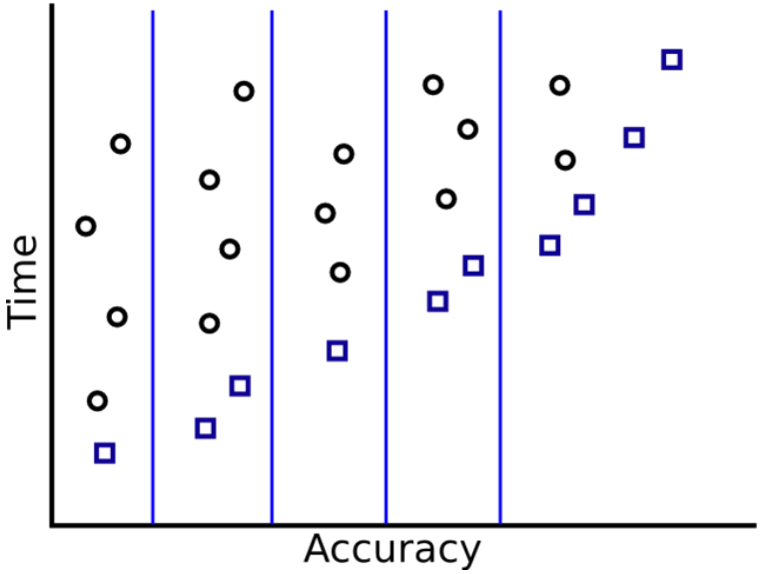
Variable accuracy autotuner



Variable accuracy autotuner

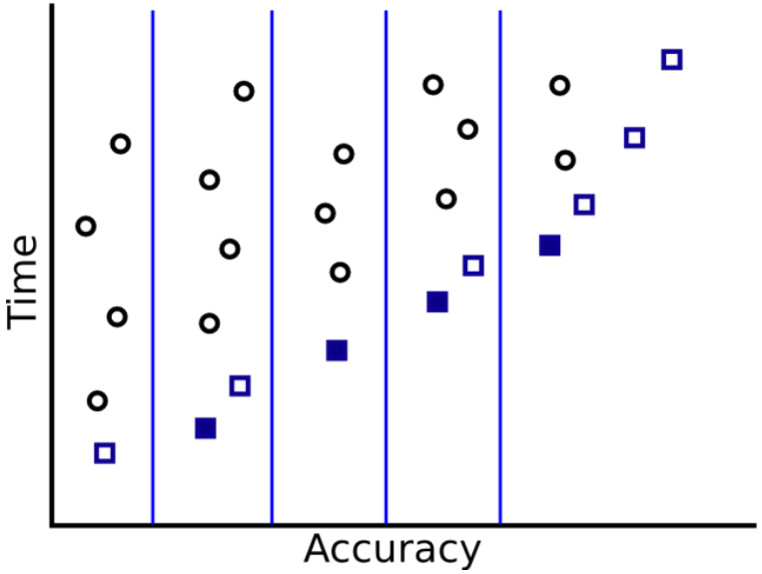


Variable accuracy autotuner



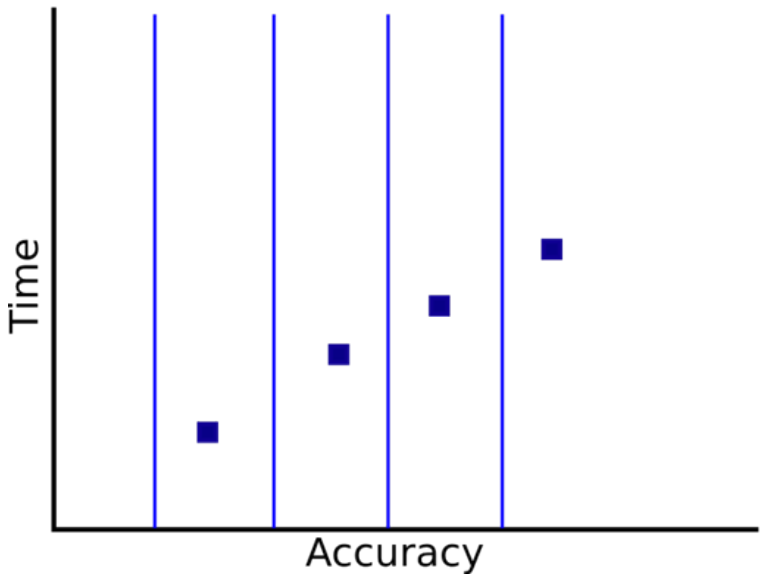
- Partition accuracy space into discrete levels

Variable accuracy autotuner



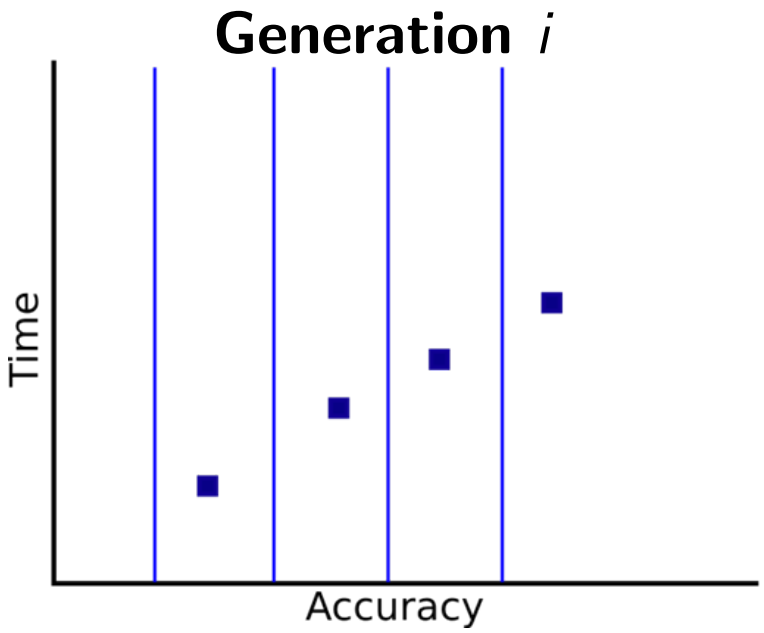
- Partition accuracy space into discrete levels
- Prune population to have a fixed number of representatives from each level

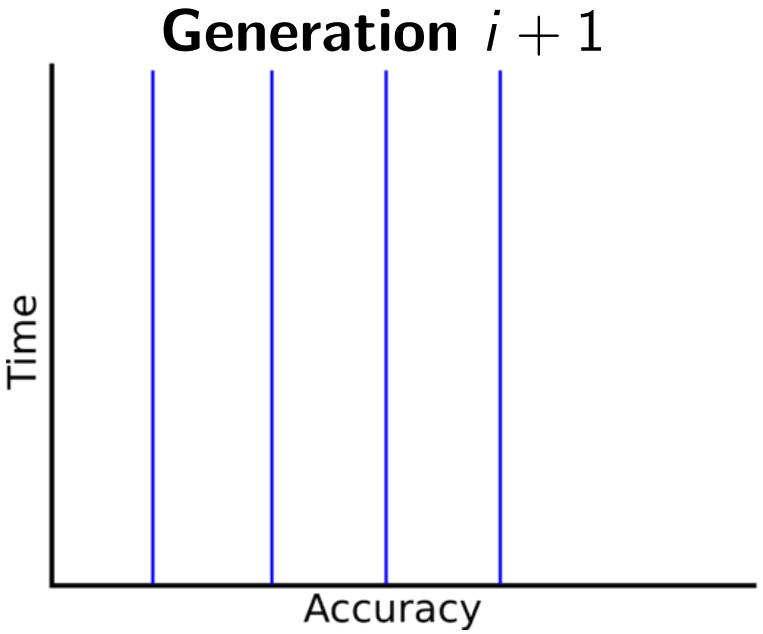
Variable accuracy autotuner



- Partition accuracy space into discrete levels
- Prune population to have a fixed number of representatives from each level

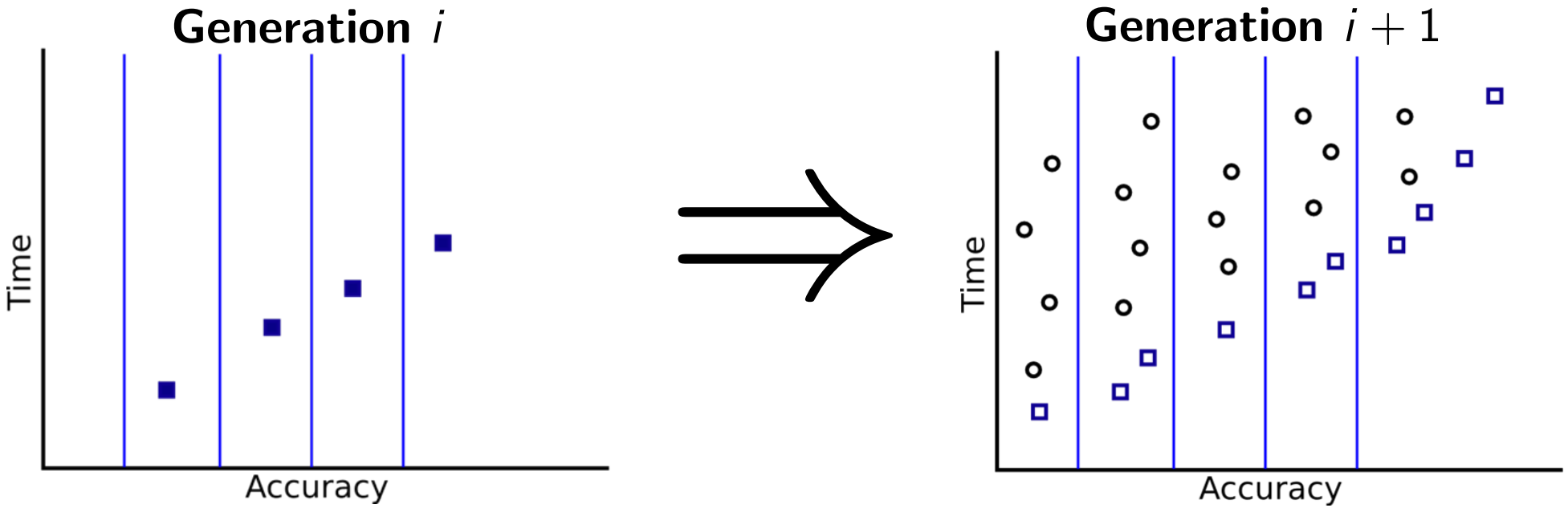
Variable accuracy autotuner





- Partition accuracy space into discrete levels
- Prune population to have a fixed number of representatives from each level

Variable accuracy autotuner



- Partition accuracy space into discrete levels
- Prune population to have a fixed number of representatives from each level

Outline

Changing accuracy requirements

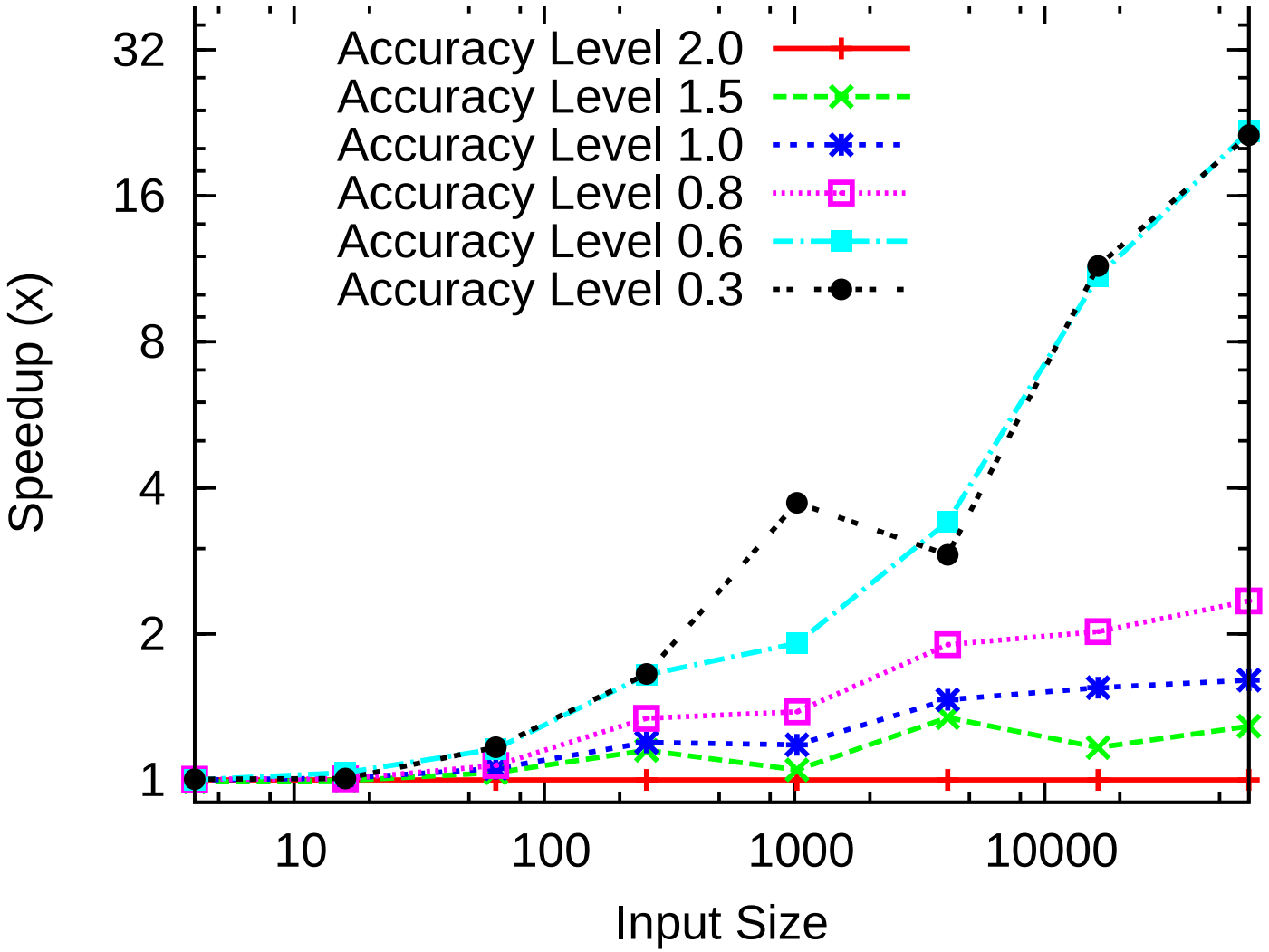
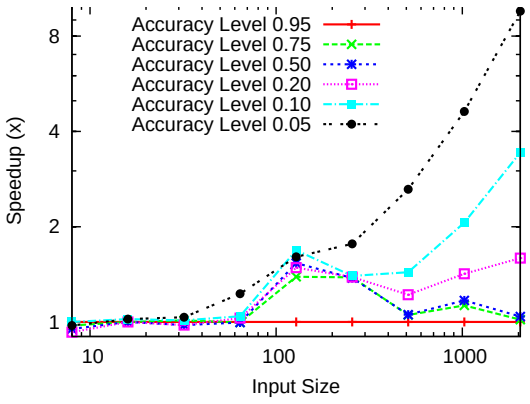
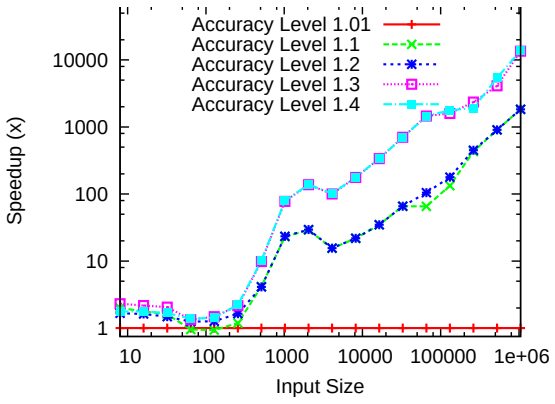


Image Compression

Changing accuracy requirements



Clustering



Bin Packing

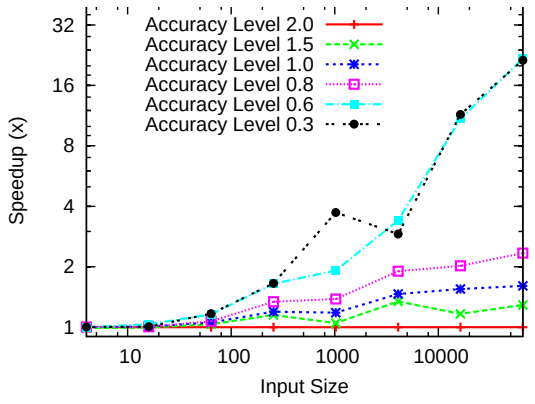
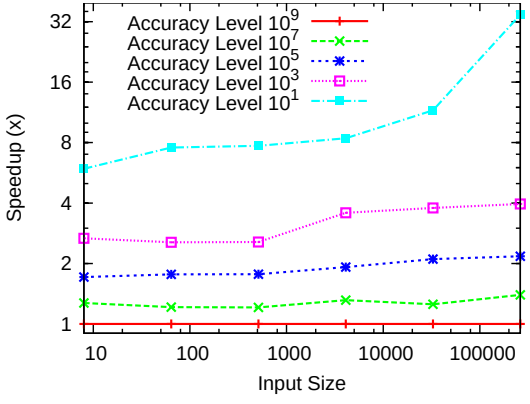
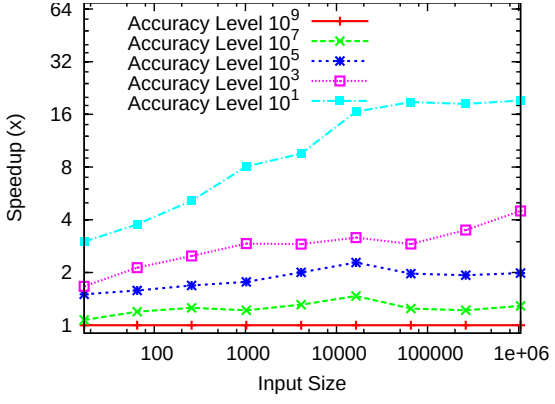


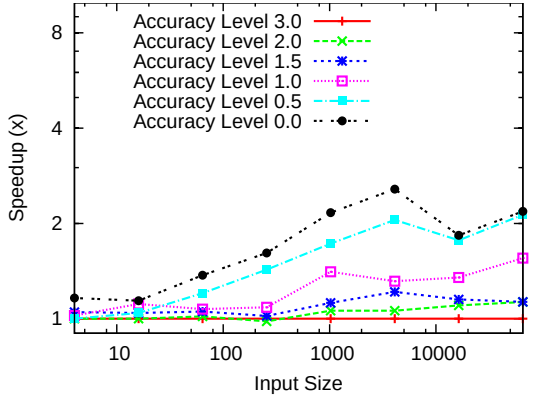
Image Compression



3D Helmholtz



2D Poisson



Preconditioner

Multigrid choice space

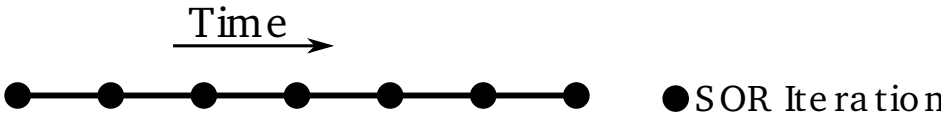
Pseudo code

```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```

Multigrid choice space

Pseudo code

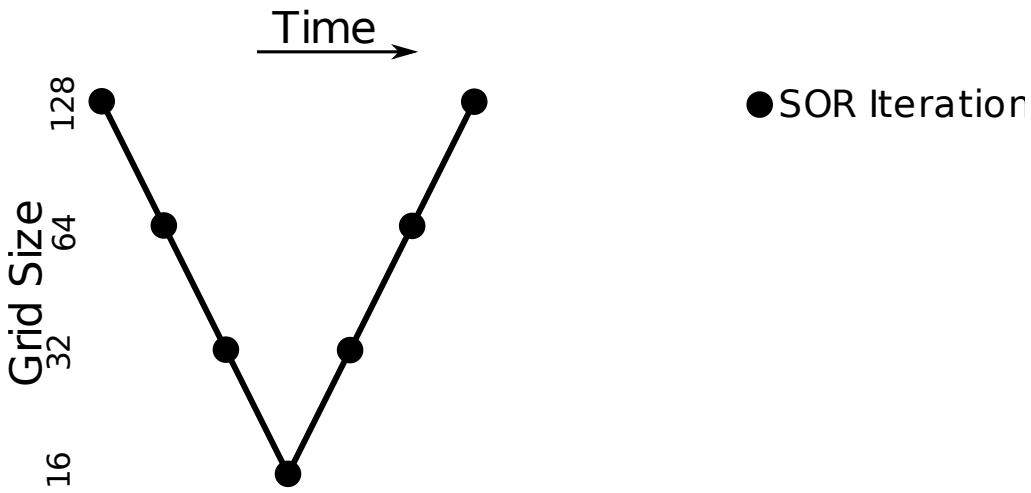
```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```



Multigrid choice space

Pseudo code

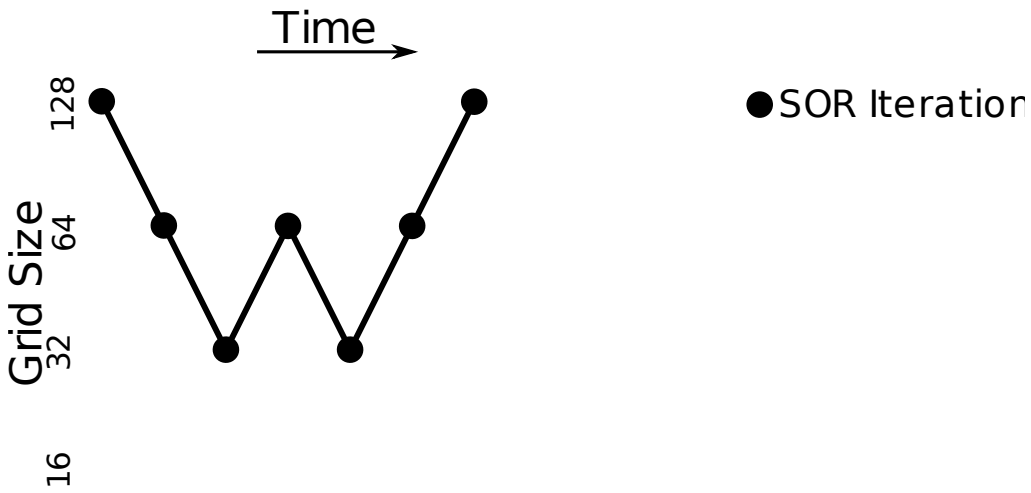
```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```



Multigrid choice space

Pseudo code

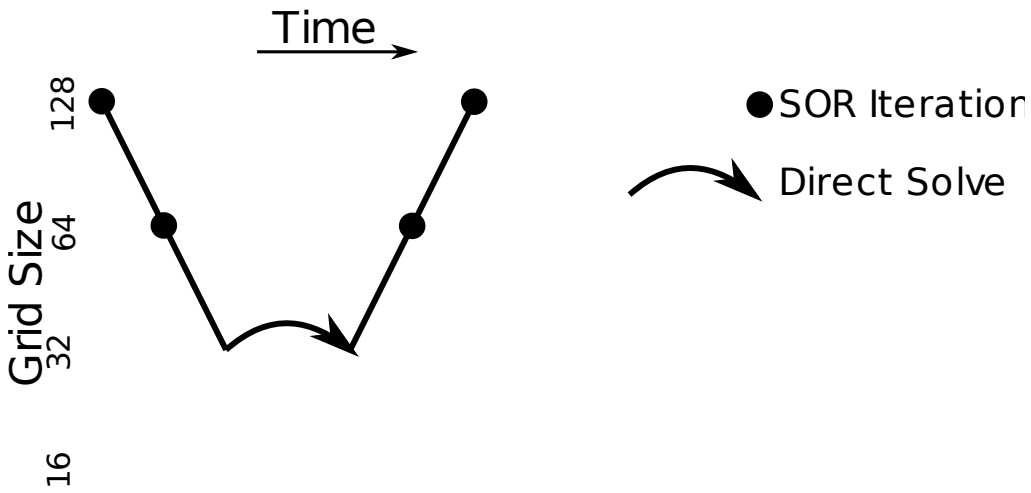
```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```



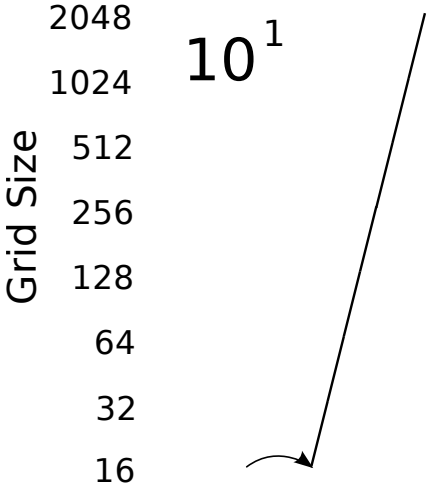
Multigrid choice space

Pseudo code

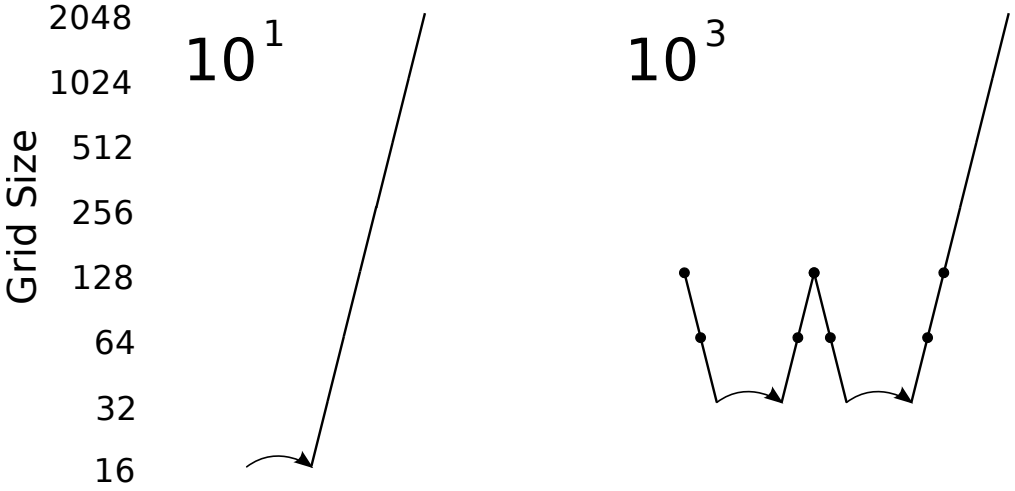
```
accuracy_metric MyRMSError
...
either {
  for_enough {
    SORIteration(tmp);
  }
} or {
  Multigrid(tmp);
} or {
  DirectSolve(tmp);
}
```



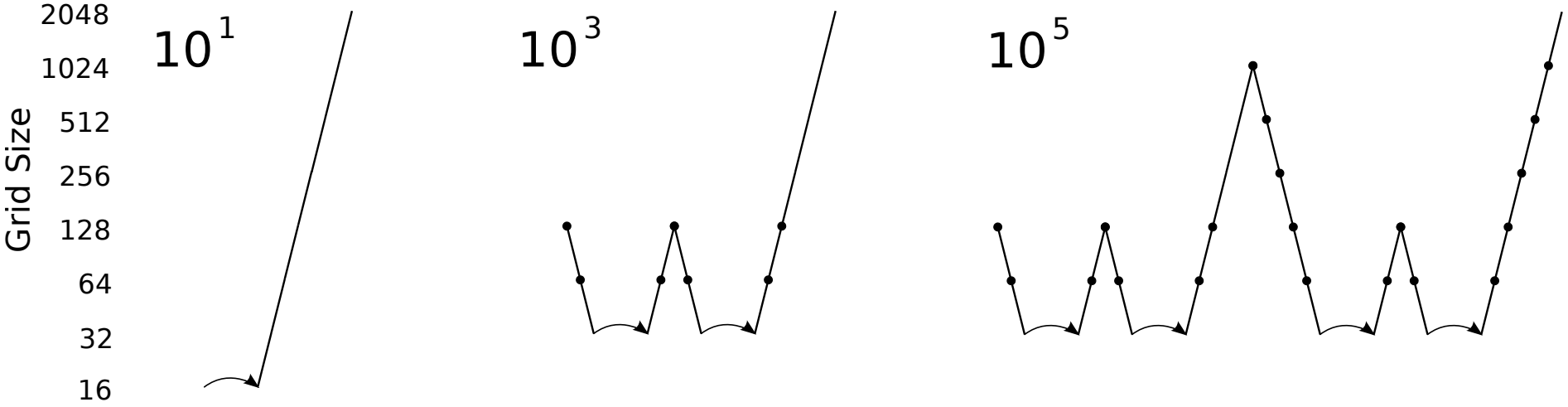
Autotuned V-cycle shapes



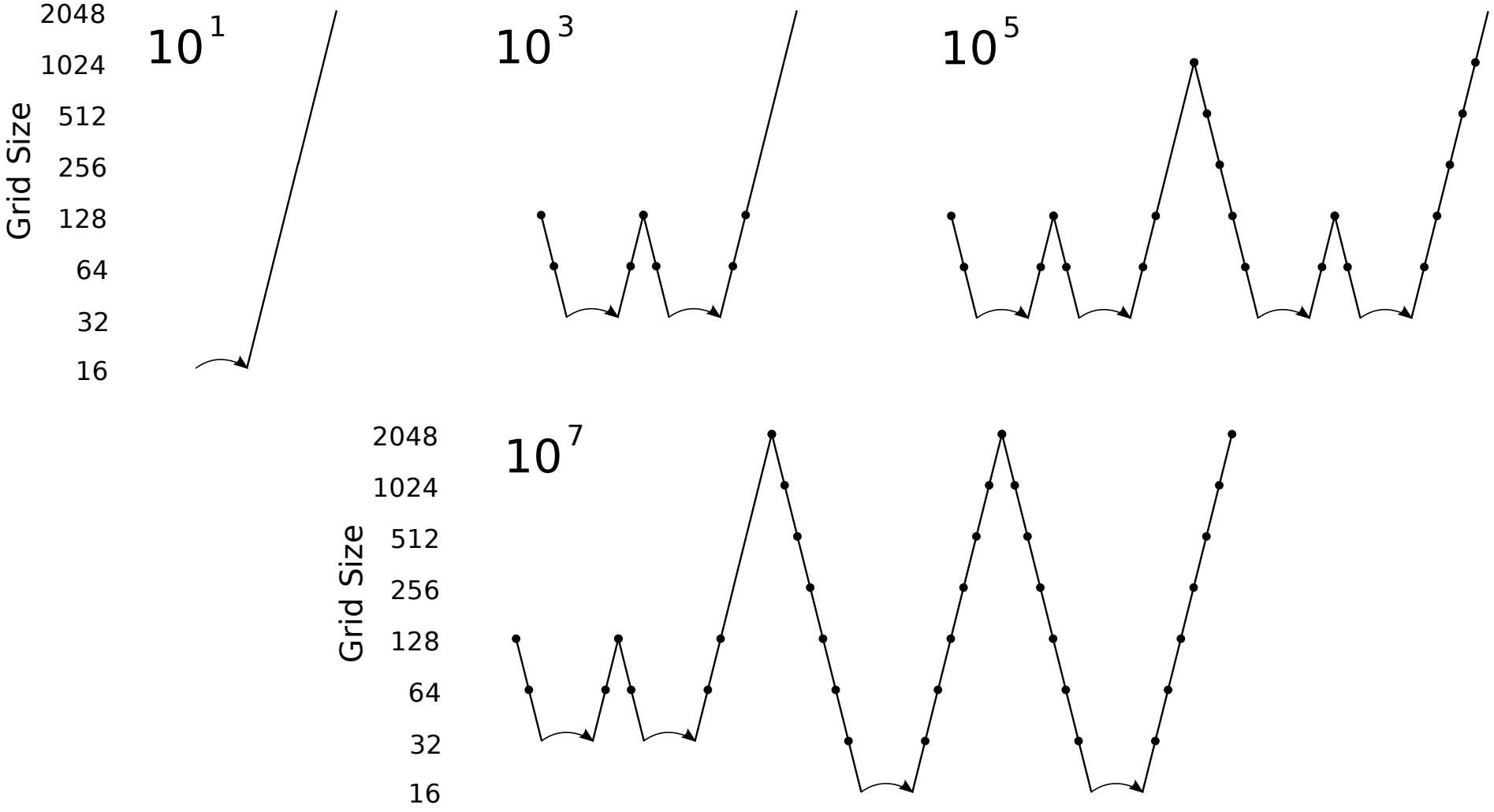
Autotuned V-cycle shapes



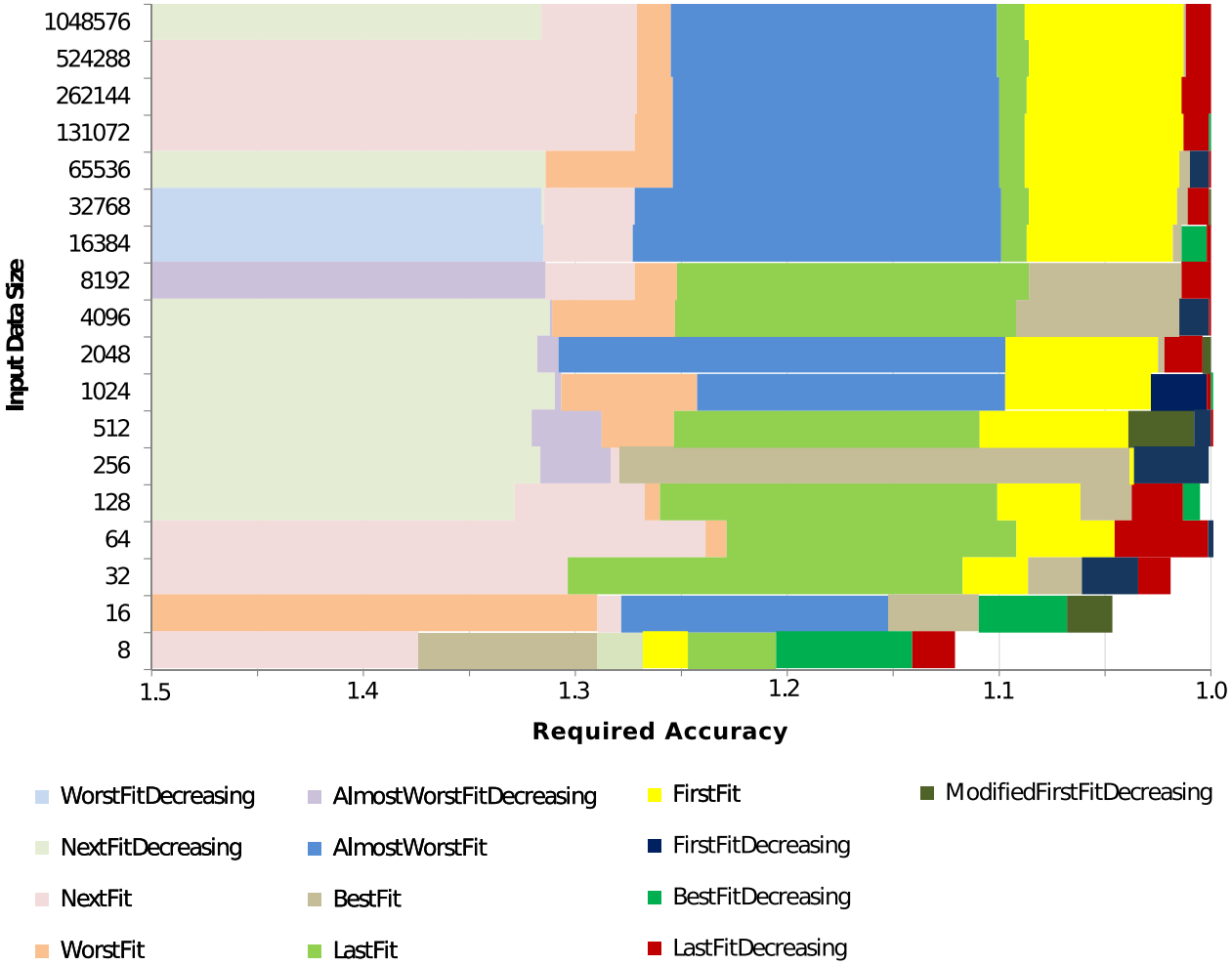
Autotuned V-cycle shapes



Autotuned V-cycle shapes



Autotuned bin packing algorithms



Outline

Conclusions

Motivating goal of PetaBricks

Make programs future-proof by allowing them to adapt to their environment.

- We can do better than hard coded constants!

Thanks!

- Questions?
- http://projects.csail.mit.edu/petabricks/