An Efficient Evolutionary Algorithm for Solving Incrementally Structured Problems

Jason Ansel Maciej Pacula
Saman Amarasinghe Una-May O'Reilly

MIT - CSAIL

July 14, 2011

Who are we?

- I do research in programming languages (PL) and compilers
- The PetaBricks language is a collaboration between:
  - A PL / compiler research group
  - A evolutionary algorithms research group
  - A applied mathematics research group

Who are we?

- I do research in programming languages (PL) and compilers
- The PetaBricks language is a collaboration between:
  - A PL / compiler research group
  - A evolutionary algorithms research group
  - A applied mathematics research group
- Our goal is to make programs run faster
- We use evolutionary algorithms to search for faster programs

Who are we?

- I do research in programming languages (PL) and compilers
- The PetaBricks language is a collaboration between:
  - A PL / compiler research group
  - A evolutionary algorithms research group
  - A applied mathematics research group
- Our goal is to make programs run faster
- We use evolutionary algorithms to search for faster programs
- The PetaBricks language defines search spaces of algorithmic choices

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

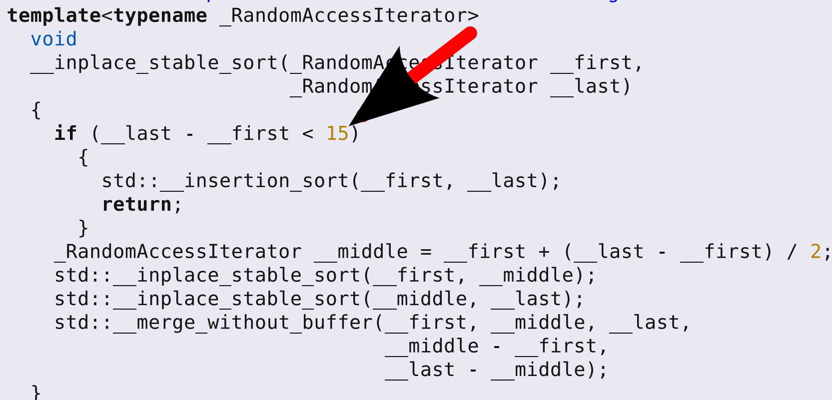
std::stable_sort

/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);
  }
```

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 $CO = 15$)

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 $CO = 15$)
- If the best cutoff has changed, perhaps best algorithm has also changed

Algorithmic choices

Language

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

Algorithmic choices

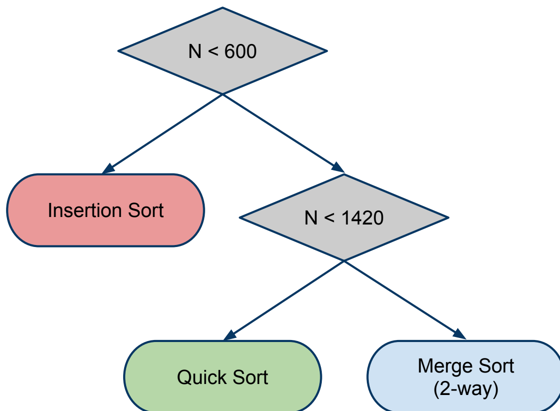
Language

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

Representation

Decision tree synthesized by our evolutionary algorithm

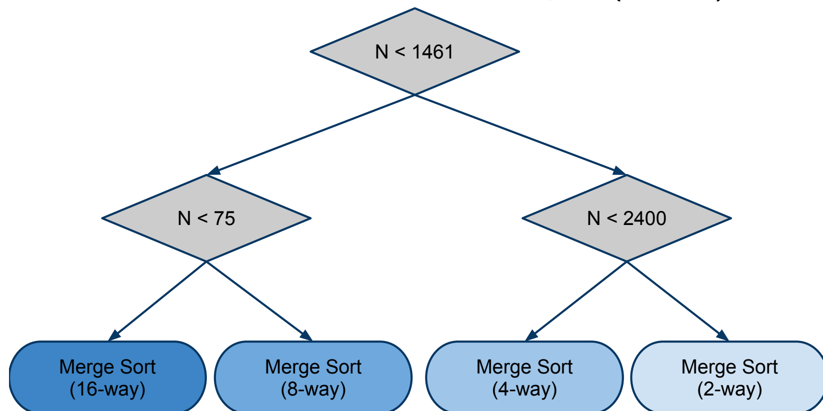
Decision trees



Text notation (will be used later): I 600 Q 1420 M^2

Decision trees

Optimized for Sun Fire T200 Niagara (8 cores):



Text notation: M^{16} 75 M^{8} 1461 M^{4} 2400 M^{2}

The configuration encoded by the genome

- Decision trees
- Algorithm parameters (integers, floats)
- Parallel scheduling / blocking parameters (integers)
- *Synthesized scalar functions* (not used in the benchmarks shown)
- The average PetaBricks benchmark's genome has:
  - 1.9 decision trees
  - 10.1 algorithm/parallelism/blocking parameters
  - 0.6 synthesized scalar functions
  - 2^{3107} possible configurations

Outline

1 PetaBricks Language

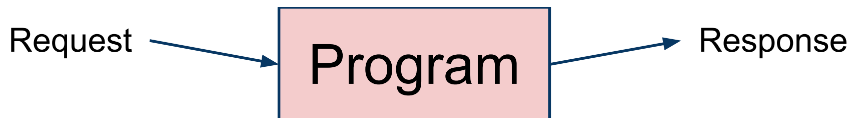
2 Autotuning Problem

3 INCREA

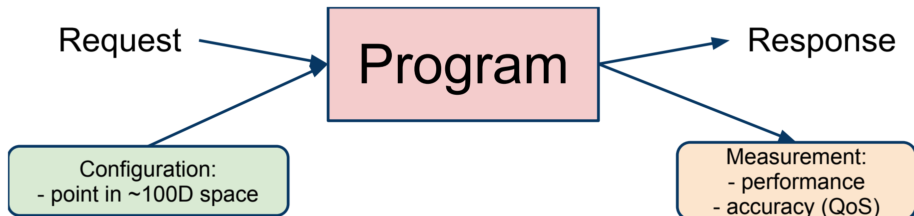
4 Evaluation

5 Conclusions

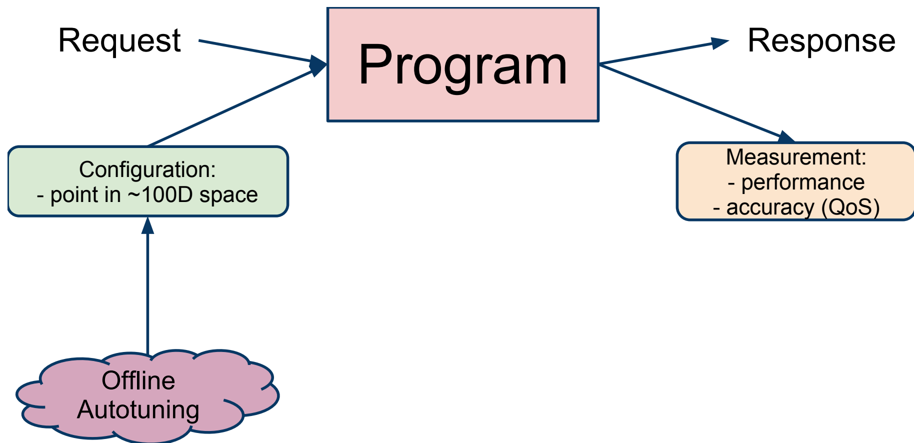
PetaBricks programs at runtime



PetaBricks programs at runtime



PetaBricks programs at runtime



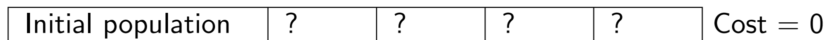
The challenges

- Evaluating objective function is expensive
  - Must run the program (at least once)
  - More expensive for unfit solutions
  - Scales poorly with larger problem sizes
- Fitness is noisy
  - Randomness from parallel races and system noise
  - Testing each candidate only once often produces an worse algorithm
  - Running many trials is expensive
- Decision tree structures are complex
  - Theoretically infinite size
  - We artificially bound them to 2^{736} bits (23 ints) each

Contrast two evolutionary approaches

- GPEA: General Purpose Evolutionary Algorithm
  - Used as a baseline
- INCREA: Incremental Evolutionary Algorithm
  - Bottom-up approach
  - Noisy fitness evaluation strategy
  - Domain informed mutation operators

General purpose evolution algorithm (GPEA)



Initial population	?	?	?	?

Cost = 0

General purpose evolution algorithm (GPEA)

Initial population	72.7s	?	?	?

Cost = 72.7

General purpose evolution algorithm (GPEA)

Initial population	72.7s	10.5s	?	?

Cost = 83.2

General purpose evolution algorithm (GPEA)

Initial population	72.7s	10.5s	4.1s	?

Cost = 87.3

General purpose evolution algorithm (GPEA)

Initial population	72.7s	10.5s	4.1s	31.2s

Cost = 118.5

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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

General purpose evolution algorithm (GPEA)

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 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

Noisy fitness evaluation

- Both strategies terminate slow tests early
- GPEA uses 1 trial per candidate algorithm
- INCREA adaptively changes the number of trials
- Represents fitness as a probability distribution
- Runs a single tailed t-test to get confidence in differences
- Runs more trails if confidence is low

Domain informed mutation operators

- Mutation operators deal with larger structures in the genome
  - "Add algorithm Y to the top of decision tree X"
  - "Scale cutoff X using a lognormal distribution"
- Generated fully automatically by our compiler

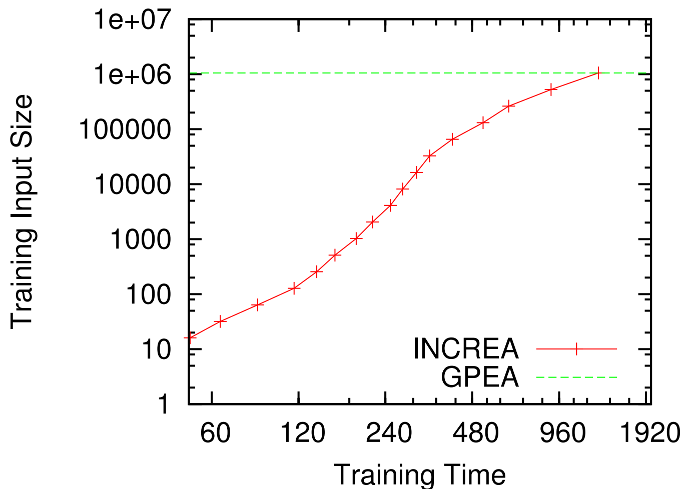
Outline

1. PetaBricks Language
2. Autotuning Problem
3. INCREA
4. Evaluation
5. Conclusions

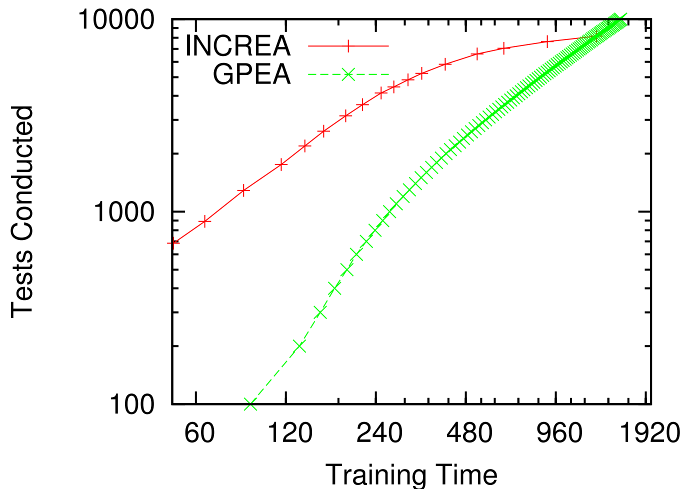
Experimental Setup

- Measuring convergence time
  - Important to both program users and developers
  - Vital in online autotuning
- Three *fixed-accuracy* PetaBricks programs:
  - Sort 2^{20} (small input size)
  - Sort 2^{23} (large input size)
  - Matrix multiply
  - Eigenvector solve
- Representative runs
- Average of 30 runs, with tests for statistical significance in paper
- Run an 8-core Xeon running Debian 5.0

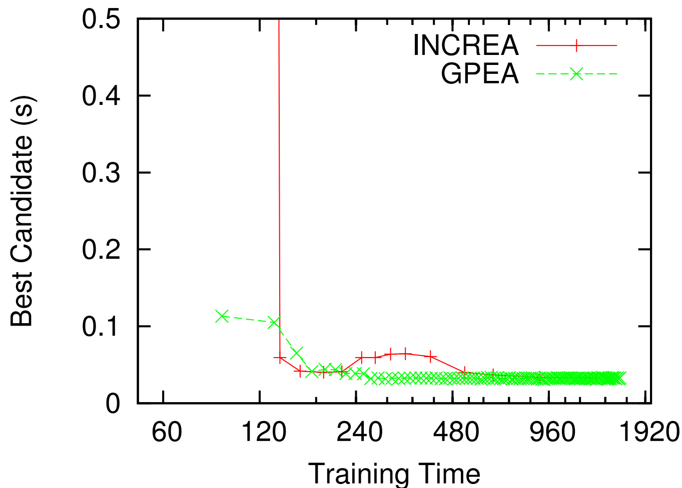
Sort 2^{20}: training input size



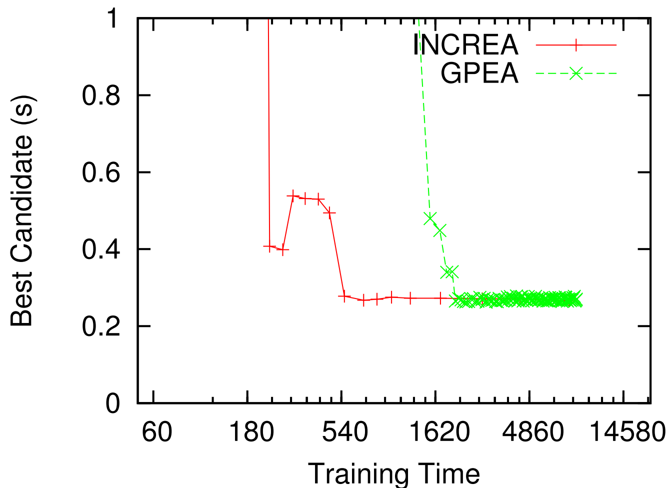
Sort 2^{20}: candidates tested



Sort 2^{20}: performance



Sort 2^{23}: performance



INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
	…	
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
...		
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
	...	
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
...		
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
...		
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

INCREA: Sort, best algorithm at each generation

Input size	Training time (s)	Genome
2^0	6.9	Q 64 Q_p
2^1	14.6	Q 64 Q_p
2^2	26.6	I
...		
2^7	115.7	I
2^8	138.6	I 270 R 1310 R_p
2^9	160.4	I 270 Q 1310 Q_p
2^{10}	190.1	I 270 Q 1310 Q_p
2^{11}	216.4	I 270 Q 3343 Q_p
2^{12}	250.0	I 189 R 13190 R_p
2^{13}	275.5	I 189 R 13190 R_p
2^{14}	307.6	I 189 R 17131 R_p
2^{15}	341.9	I 189 R 49718 R_p
2^{16}	409.3	I 189 R 124155 M^2
2^{17}	523.4	I 189 Q 5585 Q_p
2^{18}	642.9	I 189 Q 5585 Q_p
2^{19}	899.8	I 456 Q 5585 Q_p
2^{20}	1313.8	I 456 Q 5585 Q_p

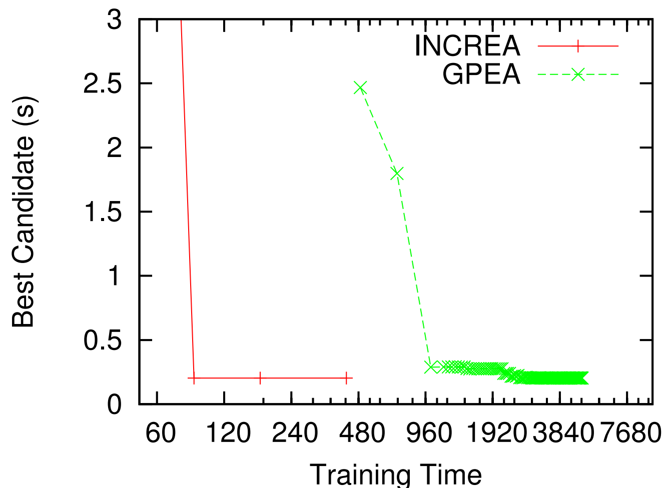
- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

GPEA: Sort, best algorithm at each generation

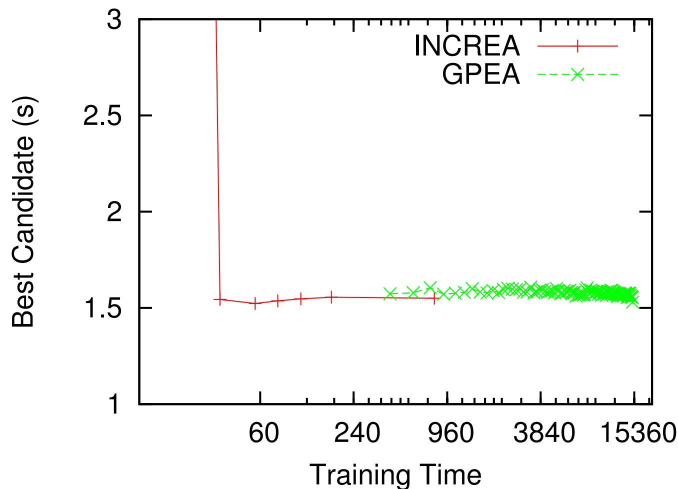
Generation	Training time (s)	Genome
0	91.4	I 448 R
1	133.2	I 413 R
2	156.5	I 448 R
3	174.8	I 448 Q
4	192.0	I 448 Q
5	206.8	I 448 Q
6	222.9	I 448 Q 4096 Q_p
7	238.3	I 448 Q 4096 Q_p
8	253.0	I 448 Q 4096 Q_p
9	266.9	I 448 Q 4096 Q_p
10	281.1	I 371 Q 4096 Q_p
11	296.3	I 272 Q 4096 Q_p
12	310.8	I 272 Q 4096 Q_p
...		
27	530.2	I 272 Q 4096 Q_p
28	545.6	I 272 Q 4096 Q_p
29	559.5	I 370 Q 8192 Q_p
30	574.3	I 370 Q 8192 Q_p
...		

- I = insertion-sort
- Q = quick-sort
- R = radix-sort
- M^x = x-way merge-sort
- $_p$ indicates run in parallel

Matrix Multiply (input size 1024x1024)



Eigenvector Solve (input size 1024x1024)



Outline

1 PetaBricks Language

2 Autotuning Problem

3 INCREA

4 Evaluation

5 Conclusions

Conclusions

Take away

The technique of solving incrementally structured problems by exploiting knowledge from smaller problem instances may be more broadly applicable.

Take away

PetaBricks is a useful framework for comparing techniques for autotuning programs.

Thanks!

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