PetaBricks: A Language and Compiler based on Autotuning

Saman Amarasinghe

Joint work with
Jason Ansel, Marek Olszewski, Cy Chan, Yee Lok Wong,
Maciej Pacula, Una-May O'Reilly and Alan Edelman

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

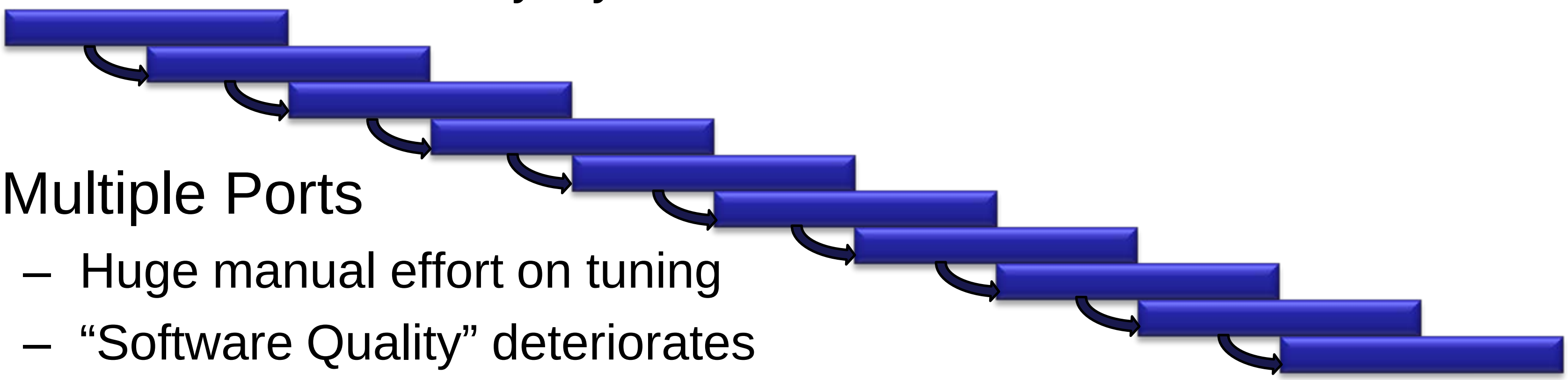
Outline

- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

Observation 1: Software Lifetime>>Hardware

- Lifetime of a software application is 30+ years

- Lifetime of a computer system is less than 6 years
  - New hardware every 3 years



- Multiple Ports
  - Huge manual effort on tuning
  - “Software Quality” deteriorates in each port
- Needs performance portability
  - Do to performance what Java did to functionality
  - Future Proofing Programs

Observation 2: Algorithmic Choice

- For many problems there are multiple algorithms
  - Most cases there is no single winner
  - An algorithm will be the best performing for a given:
    - Input size
    - Amount of parallelism
    - Communication bandwidth / synchronization cost
    - Data layout
    - Data itself (sparse data, convergence criteria etc.)
- Multicores exposes many of these to the programmer
  - Exponential growth of cores (impact of Moore’s law)
  - Wide variation of memory systems, type of cores etc.
- No single algorithm can be the best for all the cases

Observation 3: Natural Parallelism

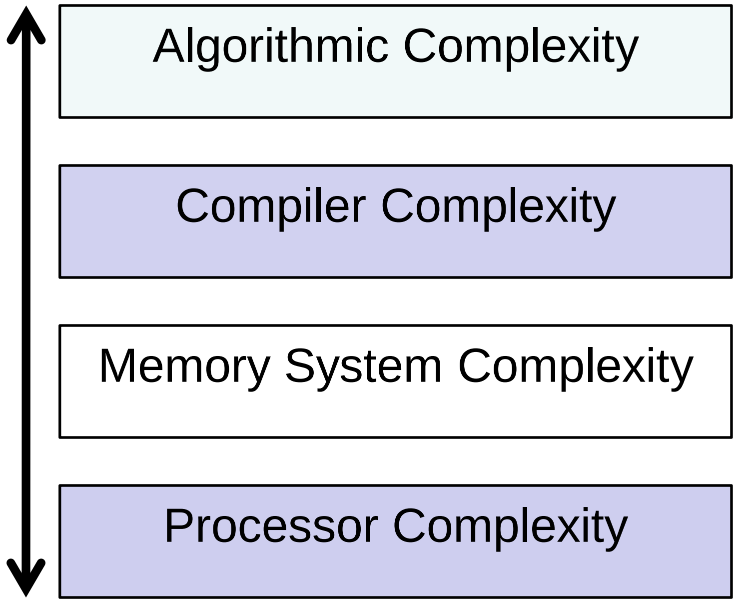
- World is a parallel place
  - It is natural to many, e.g. mathematicians
    - • , sets, simultaneous equations, etc.

- It seems that computer scientists have a hard time thinking in parallel
  - We have unnecessarily imposed sequential ordering on the world
    - Statements executed in sequence
    - for i= 1 to n
    - Recursive decomposition (given f(n) find f(n+1))

- This was useful at one time to limit the complexity….
But a big problem in the era of multicores

Observation 4: Autotuning

- Good old days → model based optimization
- Now
  - Machines are too complex to accurately model
  - Compiler passes have many subtle interactions
  - Thousands of knobs and billions of choices

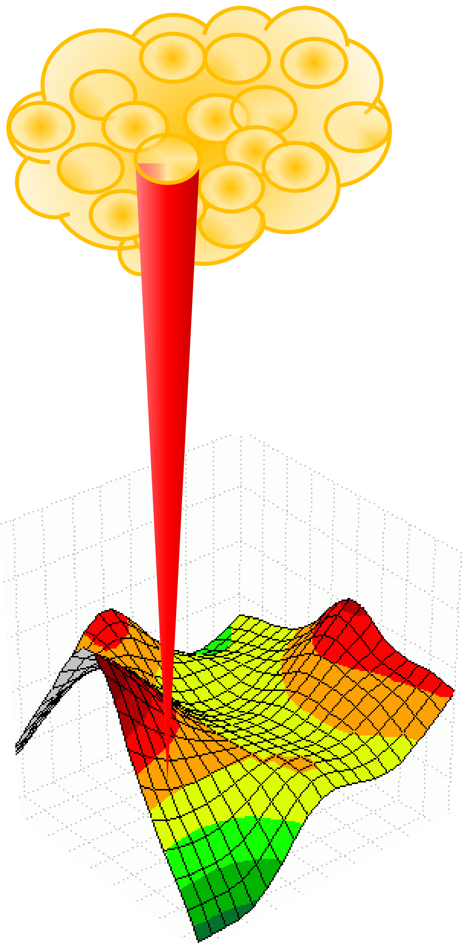


- But…
  - Computers are cheap
  - We can do end-to-end execution of multiple runs
  - Then use machine learning to find the best choice

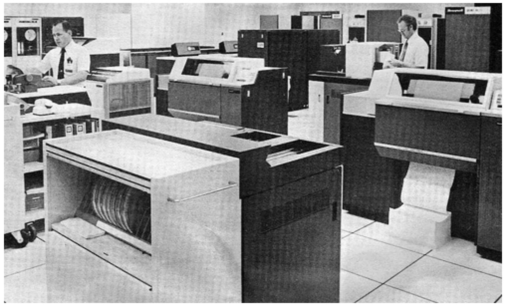
Outline

- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

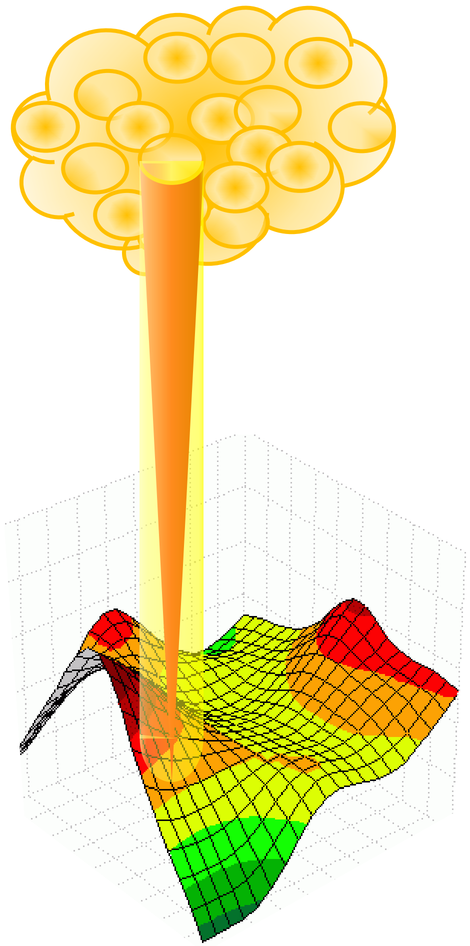
Ancient Days…



- Computers had limited power
- Compiling was a daunting task
- Languages helped by limiting choice
- Overconstraint programming languages that express only a single choice of:
  - Algorithm
  - Iteration order
  - Data layout
  - Parallelism strategy

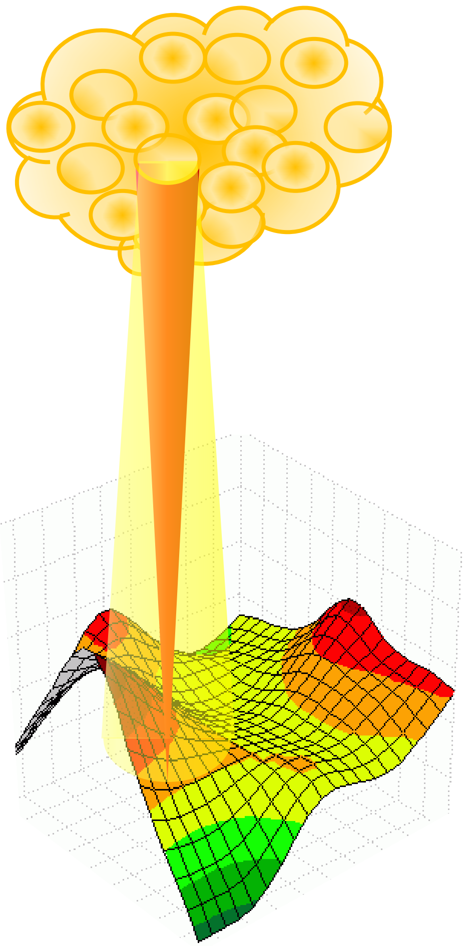


....as we progressed....



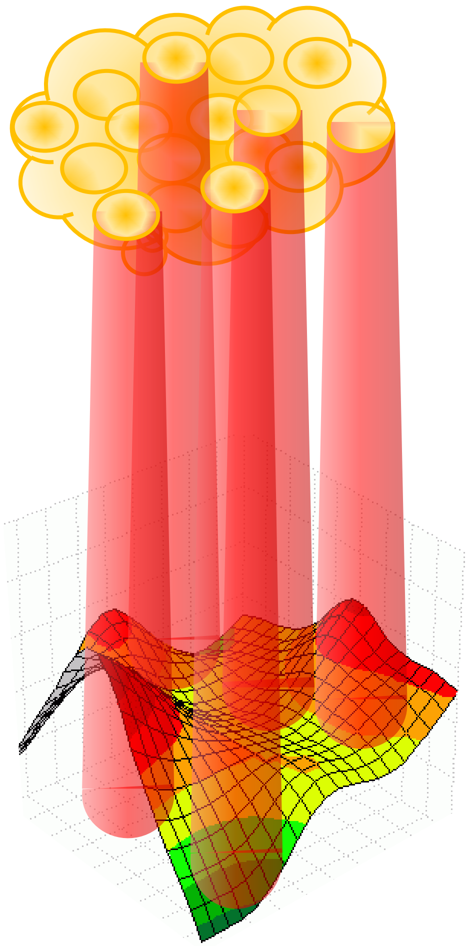
- Computers got faster
- More cycles available to the compiler
- Wanted to optimize the programs, to make them run better and faster

…and we ended up at

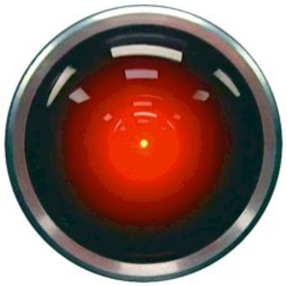


- Computers are extremely powerful
- Compilers want to do a lot
- But…the same old overconstraint languages
  - They don't provide too many choices
- Heroic analysis to rediscover some of the choices
  - Data dependence analysis
  - Data flow analysis
  - Alias analysis
  - Shape analysis
  - Interprocedural analysis
  - Loop analysis
  - Parallelization analysis
  - Information flow analysis
  - Escape analysis
  - …

Need to Rethink Languages



- Give Compiler a Choice
  - Express ‘intent’ not ‘a method’
  - Be as verbose as you can
- Muscle outpaces brain
  - Compute cycles are abundant
  - Complex logic is too hard



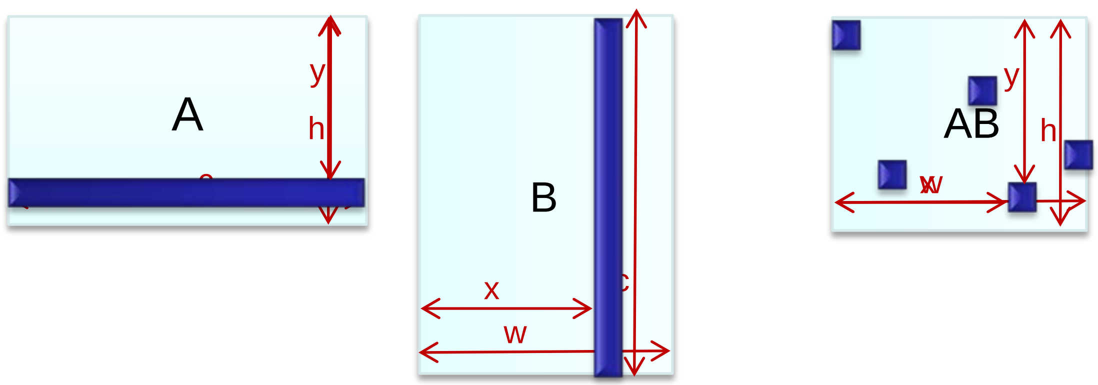
Outline

- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

PetaBricks Language

```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
  // Base case, compute a single element
  to(AB.cell(x,y) out)
  from(A.row(y) a, B.column(x) b) {
    out = dot(a, b);
  }
}
```

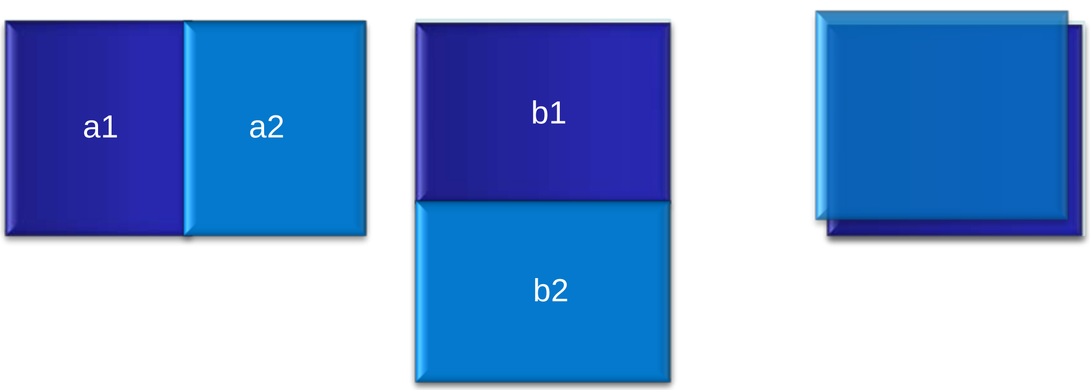
- Implicitly parallel description



PetaBricks Language

```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
   // Base case, compute a single element
   to(AB.cell(x,y) out)
   from(A.row(y) a, B.column(x) b) {
      out = dot(a, b);
  }

  // Recursively decompose in c
  to(AB ab)
  from(A.region(0,   0, c/2, h  )  a1,
         A.region(c/2, 0, c,   h  )  a2,
         B.region(0,   0, w,   c/2)  b1,
         B.region(0, c/2, w,   c  )  b2) {
   ab = MatrixAdd(MatrixMultiply(a1, b1),
                            MatrixMultiply(a2, b2));
  }
```

- Implicitly parallel description
- Algorithmic choice

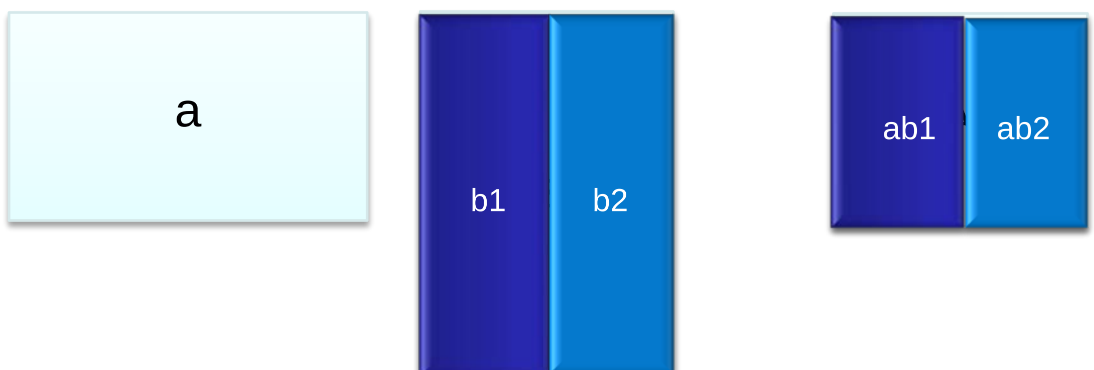


PetaBricks Language

```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
  // Base case, compute a single element
  to(AB.cell(x,y) out)
  from(A.row(y) a, B.column(x) b) {
    out = dot(a, b);
 }
```

```
// Recursively decompose in w
 to(AB.region(0,   0, w/2, h )   ab1,
    AB.region(w/2, 0, w,   h )  ab2)
 from( A a,
       B.region(0,   0, w/2, c )   b1,
       B.region(w/2, 0, w,   c )  b2) {
  ab1 = MatrixMultiply(a, b1);
  ab2 = MatrixMultiply(a, b2);
 }
```

```
 // Recursively decompose in c
 to(AB ab)
 from(A.region(0,   0, c/2, h )  a1,
      A.region(c/2, 0, c,   h )  a2,
      B.region(0,   0, w,   c/2)  b1,
      B.region(0, c/2, w,   c )  b2) {
  ab = MatrixAdd(MatrixMultiply(a1, b1),
                 MatrixMultiply(a2, b2));
 }
```



PetaBricks Language

```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
   // Base case, compute a single element
   to(AB.cell(x,y) out)
   from(A.row(y) a, B.column(x) b) {
      out = dot(a, b);
  }

  // Recursively decompose in c
  to(AB ab)
  from(A.region(0,   0, c/2, h  )  a1,
       A.region(c/2, 0, c,   h  )  a2,
       B.region(0,   0, w,   c/2)  b1,
       B.region(0, c/2, w,   c  )  b2) {
   ab = MatrixAdd(MatrixMultiply(a1, b1),
                  MatrixMultiply(a2, b2));
  }

 // Recursively decompose in w
  to(AB.region(0,   0, w/2, h  )   ab1,
     AB.region(w/2, 0, w,   h  )  ab2)
  from( A a,
        B.region(0,   0, w/2, c  )   b1,
        B.region(w/2, 0, w,   c  )  b2) {
   ab1 = MatrixMultiply(a, b1);
   ab2 = MatrixMultiply(a, b2);
  }

  // Recursively decompose in h
  to(AB.region(0, 0,   w, h/2)   ab1,
     AB.region(0, h/2, w, h  )   ab2)
  from(A.region(0,   0, c,   h/2)  a1,
       A.region(0, h/2, c,   h  )  a2,
       B b) {
   ab1=MatrixMultiply(a1, b);
   ab2=MatrixMultiply(a2, b);
  }
}
```

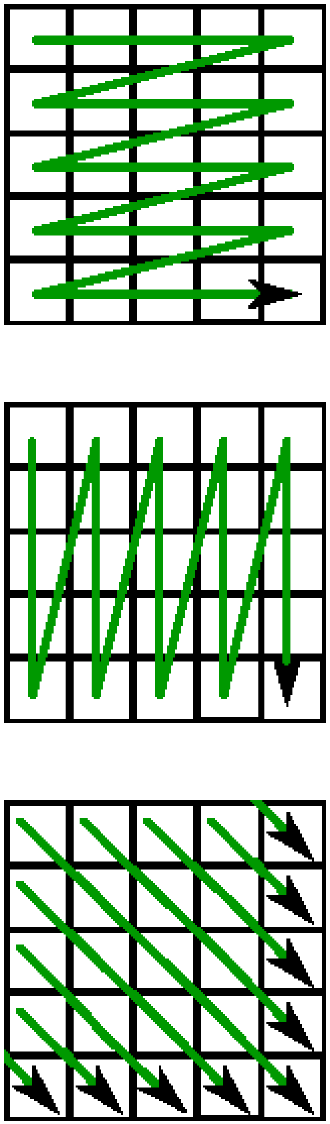
PetaBricks Language

```
transform Strassen
    from A11[n,n], A12[n,n], A21[n,n], A22[n,n],
      B11[n,n], B12[n,n], B21[n,n], B22[n,n]
    through M1[n,n], M2[n,n], M3[n,n], M4[n,n], M5[n,n], M6[n,n], M7[n,n]
    to   C11[n,n], C12[n,n], C21[n,n], C22[n,n]
    {
     to(M1 m1) from(A11 a11, A22 a22, B11 b11, B22 b22) using(t1[n,n], t2[n,n]) {
       MatrixAdd(t1, a11, a22);
       MatrixAdd(t2, b11, b22);
       MatrixMultiplySqr(m1, t1, t2);
     }
     to(M2 m2) from(A21 a21, A22 a22, B11 b11) using(t1[n,n]) {
       MatrixAdd(t1, a21, a22);
       MatrixMultiplySqr(m2, t1, b11);
     }
     to(M3 m3) from(A11 a11, B12 b12, B22 b22) using(t1[n,n]) {
       MatrixSub(t2, b12, b22);
       MatrixMultiplySqr(m3, a11, t2);
     }
     to(M4 m4) from(A22 a22, B21 b21, B11 b11) using(t1[n,n]) {
       MatrixSub(t2, b21, b11);
       MatrixMultiplySqr(m4, a22, t2);
     }
     to(M5 m5) from(A11 a11, A12 a12, B22 b22) using(t1[n,n]) {
       MatrixAdd(t1, a11, a12);
       MatrixMultiplySqr(m5, t1, b22);
     }
     to(M6 m6) from(A21 a21, A11 a11, B11 b11, B12 b12) using(t1[n,n], t2[n,n]) {
       MatrixSub(t1, a21, a11);
       MatrixAdd(t2, b11, b12);
       MatrixMultiplySqr(m6, t1, t2);
     }
     to(M7 m7) from(A12 a12, A22 a22, B21 b21, B22 b22) using(t1[n,n], t2[n,n]) {
       MatrixSub(t1, a12, a22);
       MatrixAdd(t2, b21, b22);
       MatrixMultiplySqr(m7, t1, t2);
     }

     to(C11 c11) from(M1 m1, M4 m4, M5 m5, M7 m7){
       MatrixAddAddSub(c11, m1, m4, m7, m5);
     }
     to(C12 c12) from(M3 m3, M5 m5){
       MatrixAdd(c12, m3, m5);
     }
     to(C21 c21) from(M2 m2, M4 m4){
       MatrixAdd(c21, m2, m4);
     }
     to(C22 c22) from(M1 m1, M2 m2, M3 m3, M6 m6){
       MatrixAddAddSub(c22, m1, m3, m6, m2);
     }
    }
```

Language Support for Algorithmic Choice

- Algorithmic choice is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
- Compiler re-use rules to create hybrid algorithms
- Can express choices at many different granularities

Synthesized Outer Control Flow



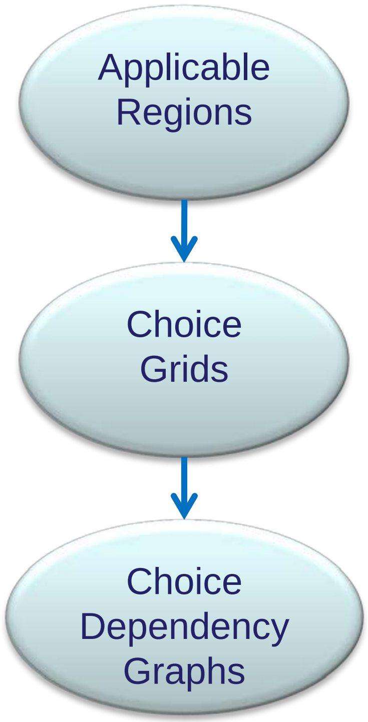
- Outer control flow synthesized by compiler
- Another choice that the programmer should not make
  - By rows?
  - By columns?
  - Diagonal? Reverse order? Blocked?
  - Parallel?
- Instead programmer provides explicit producer-consumer relations
- Allows compiler to explore choice space

Outline

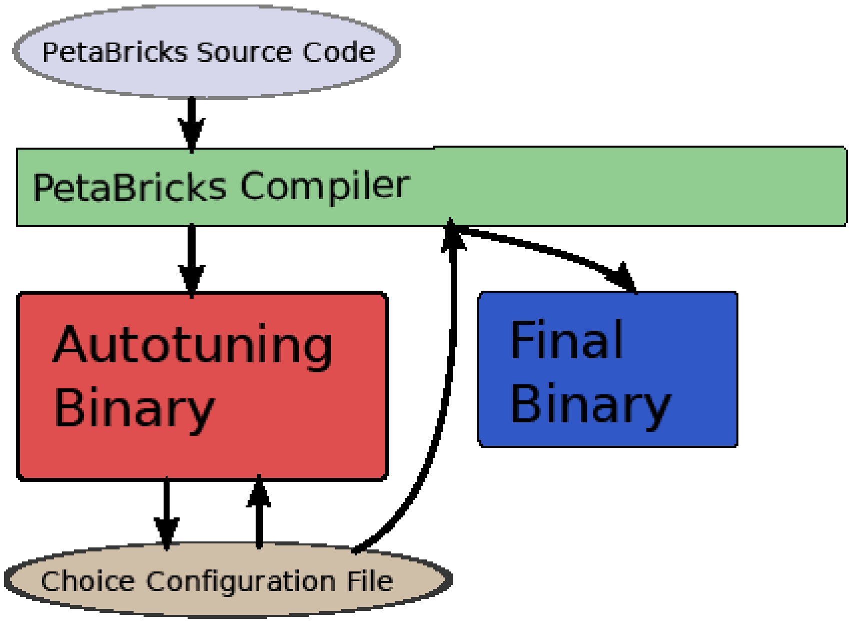
- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

Compilation Process

- Applicable Regions
- Choice Grids
- Choice Dependency Graphs



PetaBricks Flow



1. PetaBricks source code is compiled
2. An autotuning binary is created
3. Autotuning occurs creating a choice configuration file
4. Choices are fed back into the compiler to create a static binary

Autotuning

- Based on two building blocks:
  - A genetic tuner
  - An n-ary search algorithm
- Flat parameter space
- Compiler generates a dependency graph describing this parameter space
- Entire program tuned from bottom up

Outline

- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

Algorithmic Choice in Sorting

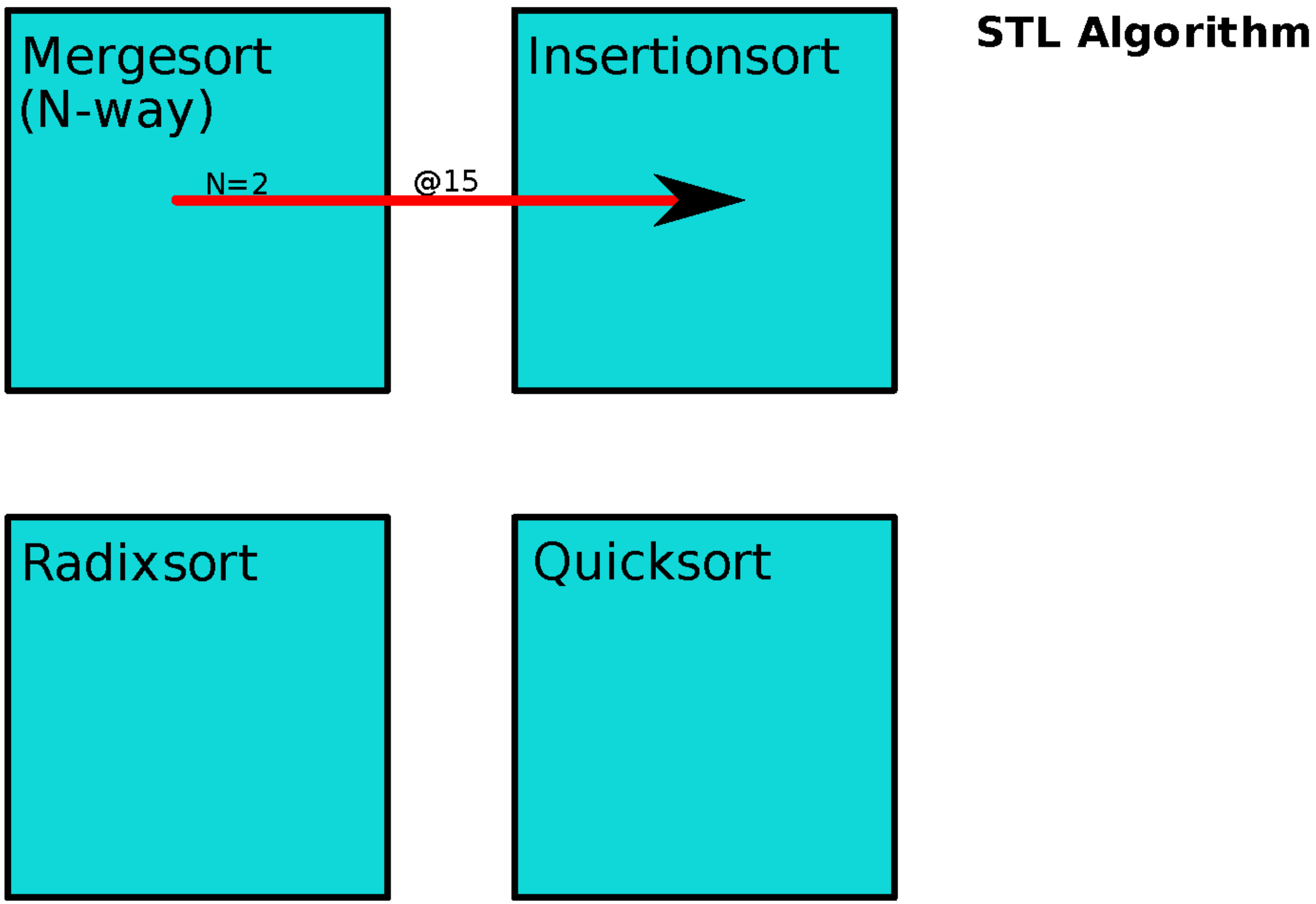
Mergesort (N-way)

Insertionsort

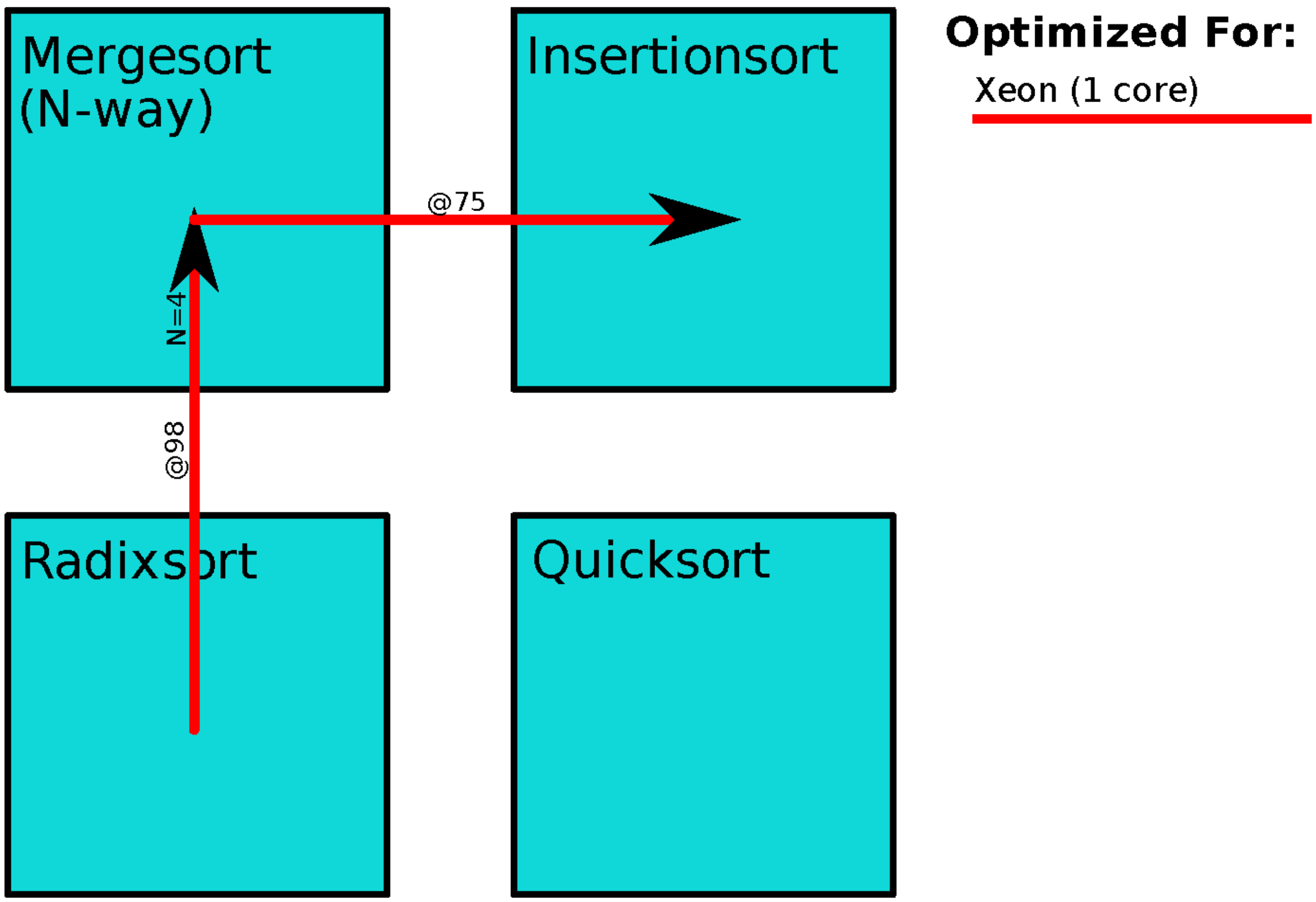
Radixsort

Quicksort

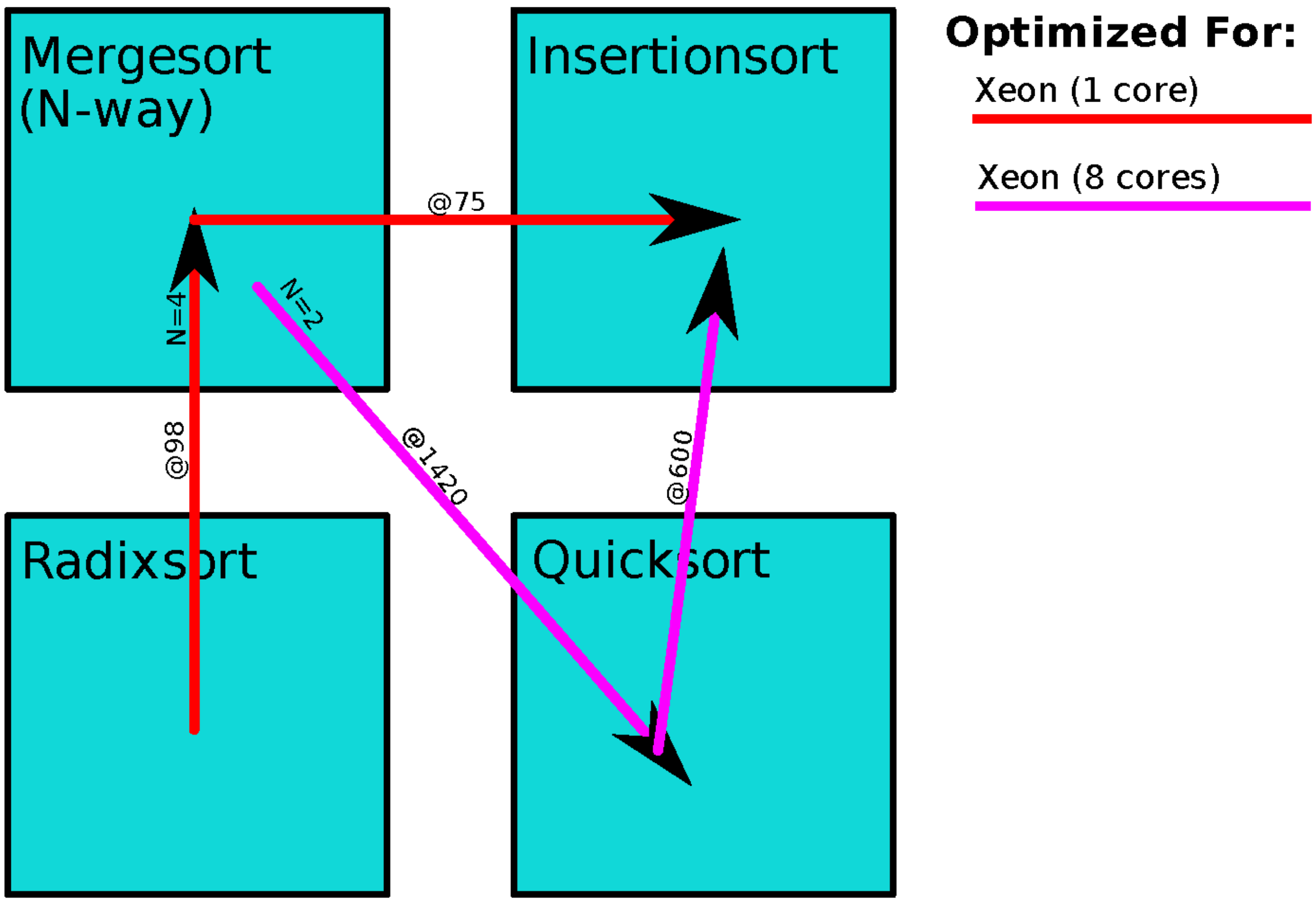
Algorithmic Choice in Sorting



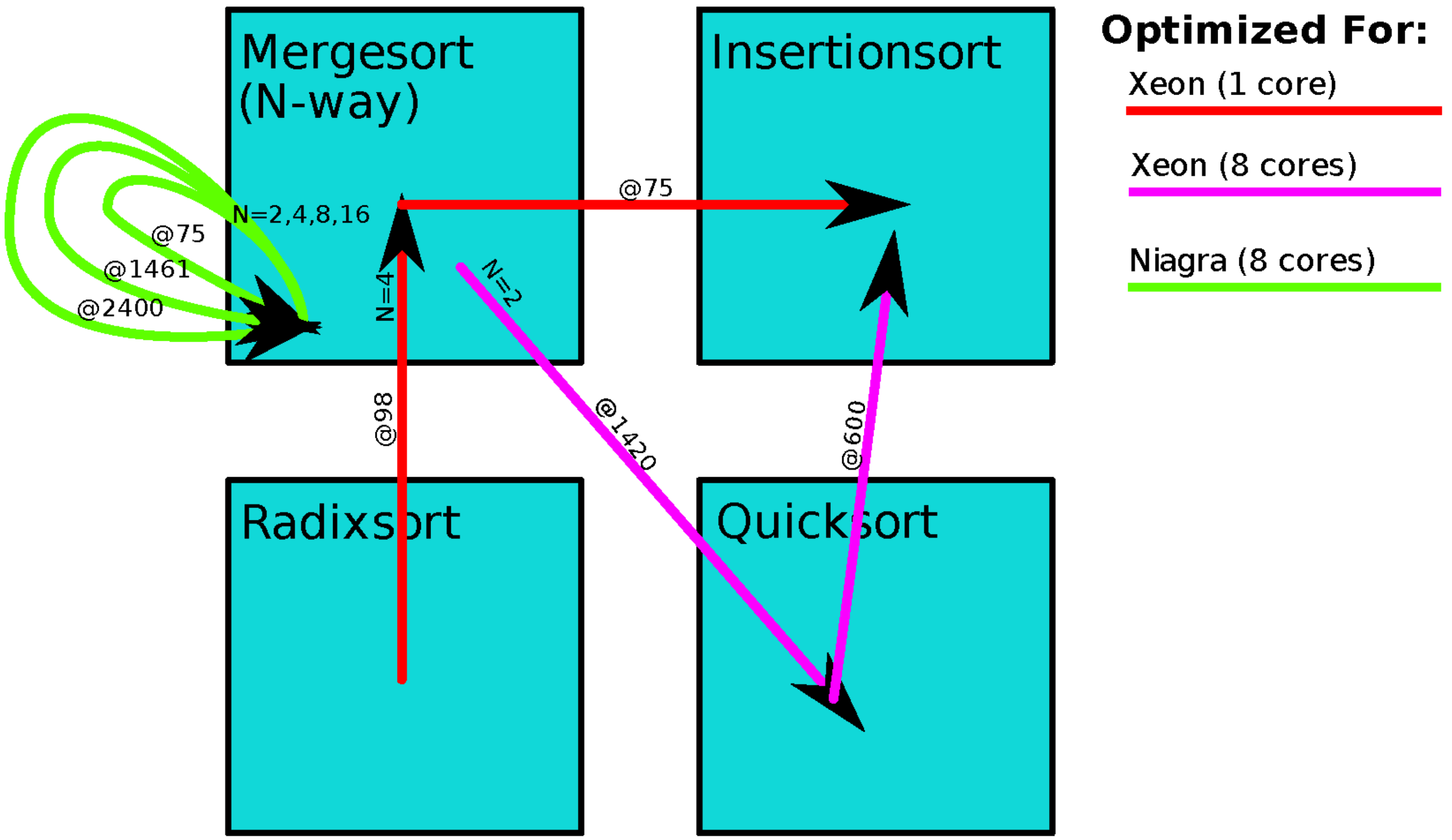
Algorithmic Choice in Sorting



Algorithmic Choice in Sorting



Algorithmic Choice in Sorting



Future Proofing Sort

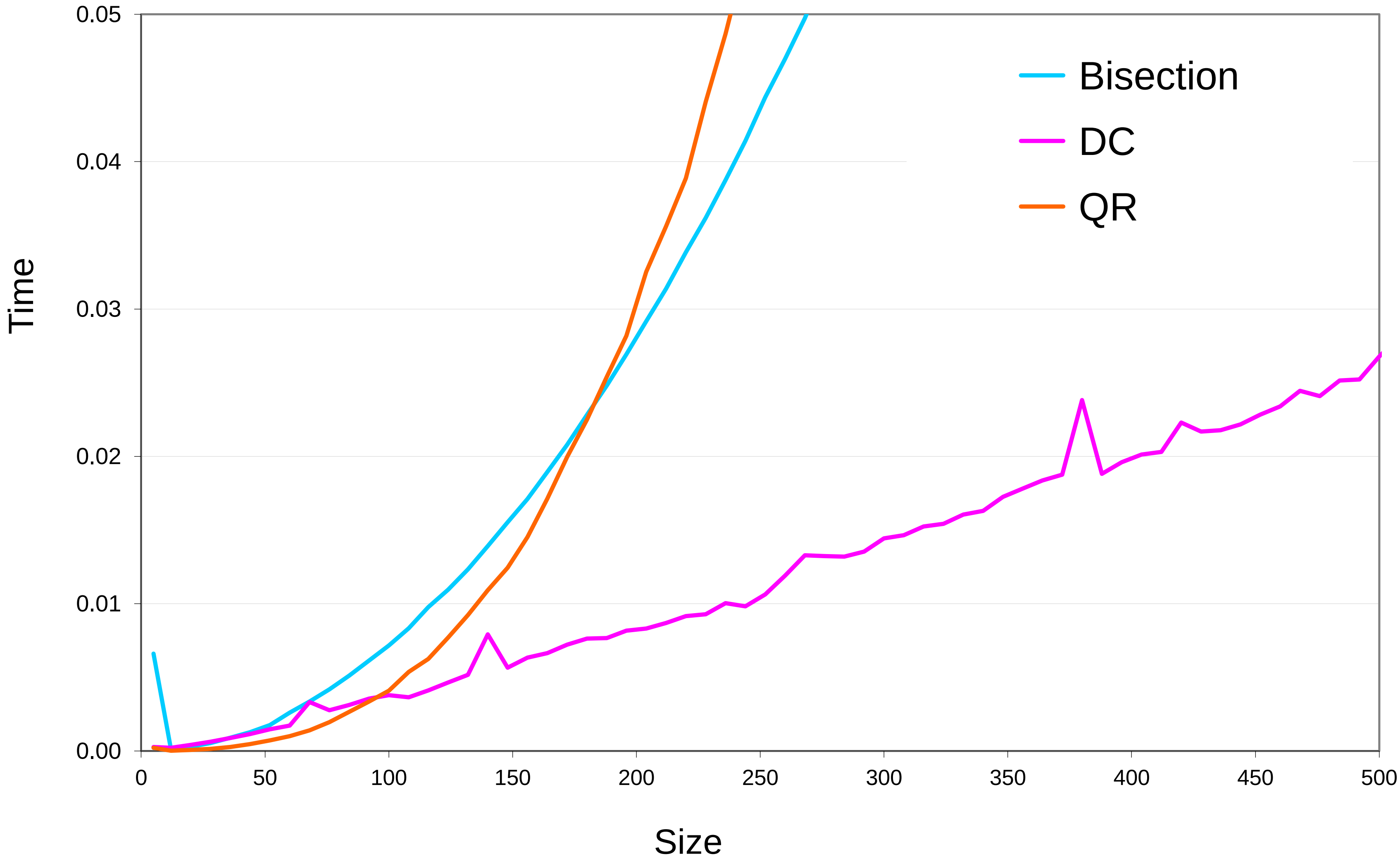
System		Cores used	Scalability	Algorithm Choices (w/ switching points)
Mobile	Core 2 Duo Mobile	2 of 2	1.92	IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(•)
Xeon 1-way	Xeon E7340 (2 x 4 core)	1 of 8	-	IS(75) 4MS(98) RS(•)
Xeon 8-way	Xeon E7340 (2 x 4 core)	8 of 8	5.69	IS(600) QS(1420) 2MS(•)
Niagara	Sun Fire T200	8 of 8	7.79	16MS(75) 8MS(1461) 4MS(2400) 2MS(•)

Future Proofing Sort

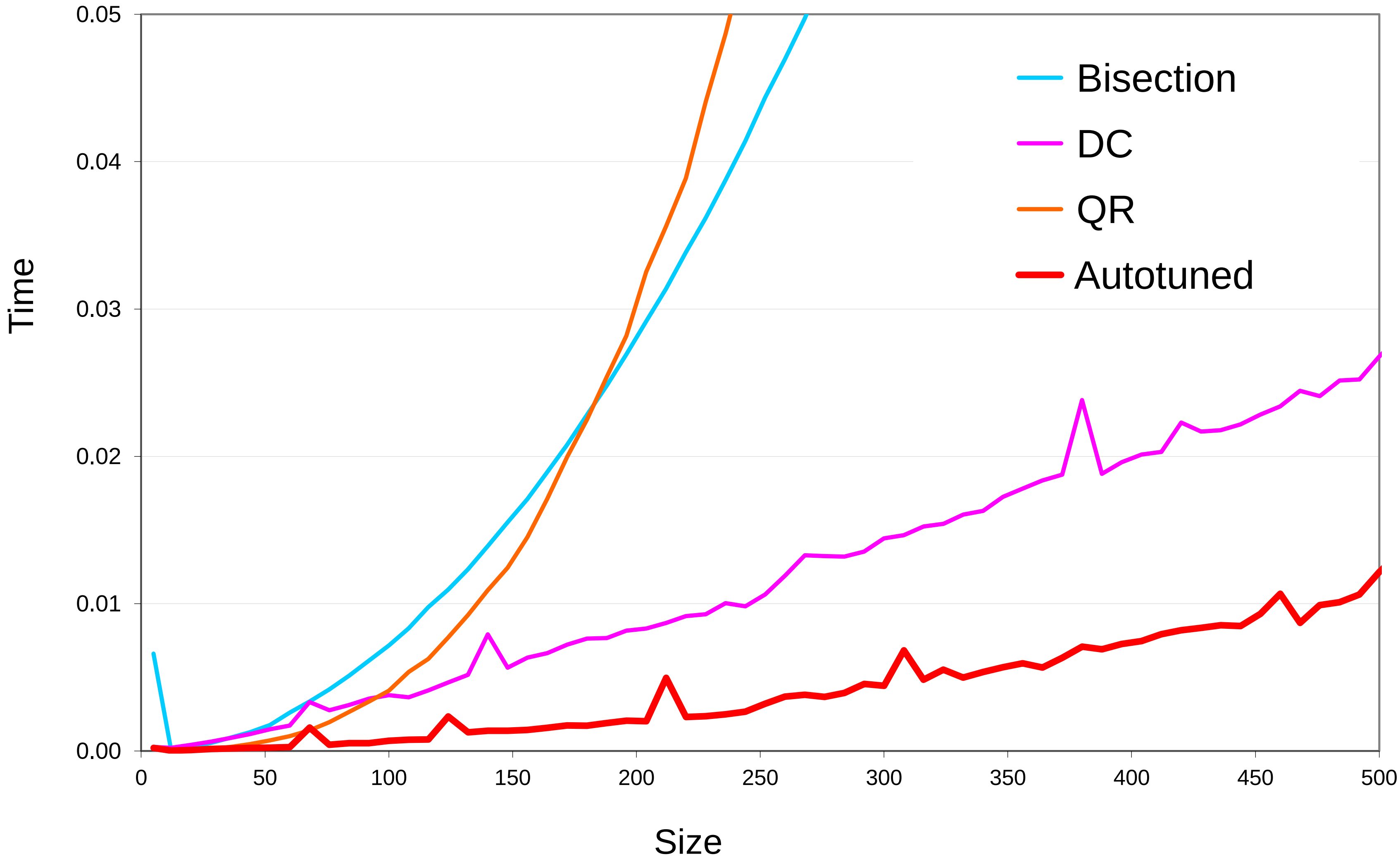
System		Cores used	Scalability	Algorithm Choices (w/ switching points)
Mobile	Core 2 Duo Mobile	2 of 2	1.92	IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(•)
Xeon 1-way	Xeon E7340 (2 x 4 core)	1 of 8	-	IS(75) 4MS(98) RS(•)
Xeon 8-way	Xeon E7340 (2 x 4 core)	8 of 8	5.69	IS(600) QS(1420) 2MS(•)
Niagara	Sun Fire T200	8 of 8	7.79	16MS(75) 8MS(1461) 4MS(2400) 2MS(•)

		Trained On			
		Mobile	Xeon 1-way	Xeon 8-way	Niagara
Run On	Mobile	-	1.09x	1.67x	1.47x
	Xeon 1-way	1.61x	-	2.08x	2.50x
	Xeon 8-way	1.59x	2.14x	-	2.35x
	Niagara	1.12x	1.51x	1.08x	-

Eigenvector Solve



Eigenvector Solve



Outline

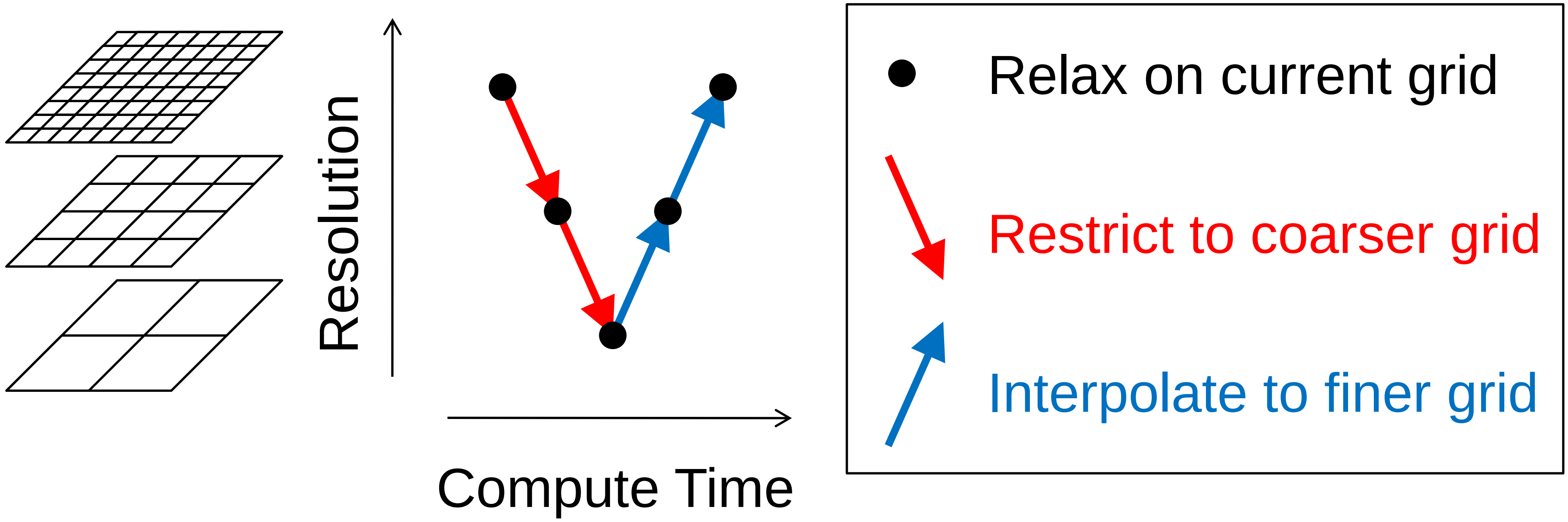
- Four Observations
- Evolution of Programming Languages
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision

Variable Accuracy Algorithms

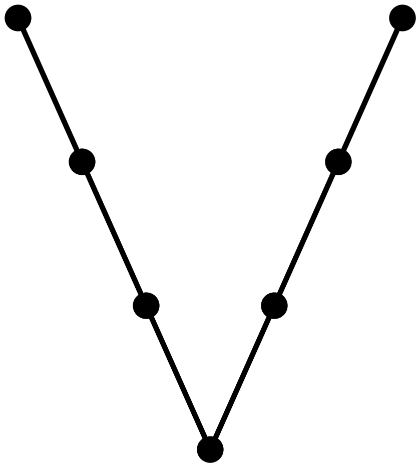
- Lots of algorithms where the accuracy of output can be tuned:
  - Iterative algorithms (e.g. solvers, optimization)
  - Signal processing (e.g. images, sound)
  - Approximation algorithms
- Can trade accuracy for speed
- All user wants: Solve to a certain accuracy as fast as possible using whatever algorithms necessary!

A Very Brief Multigrid Intro

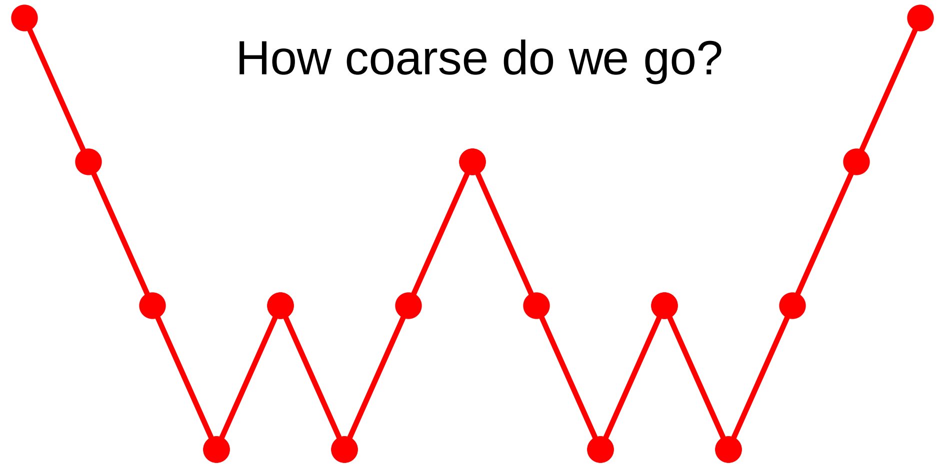
- Used to iteratively solve PDEs over a gridded domain
- **Relaxations** update points using neighboring values (stencil computations)
- **Restrictions** and **Interpolations** compute new grid with coarser or finer discretization



Multigrid Cycles

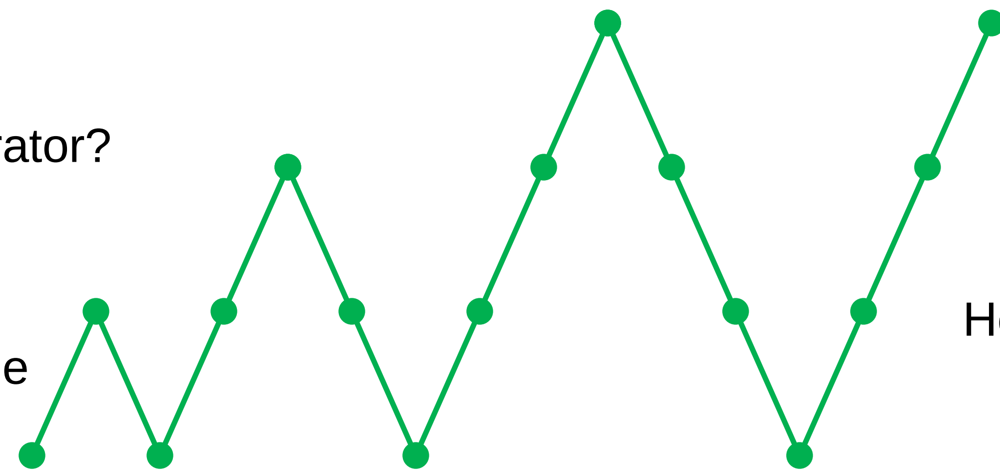


V-Cycle



W-Cycle

Relaxation operator?



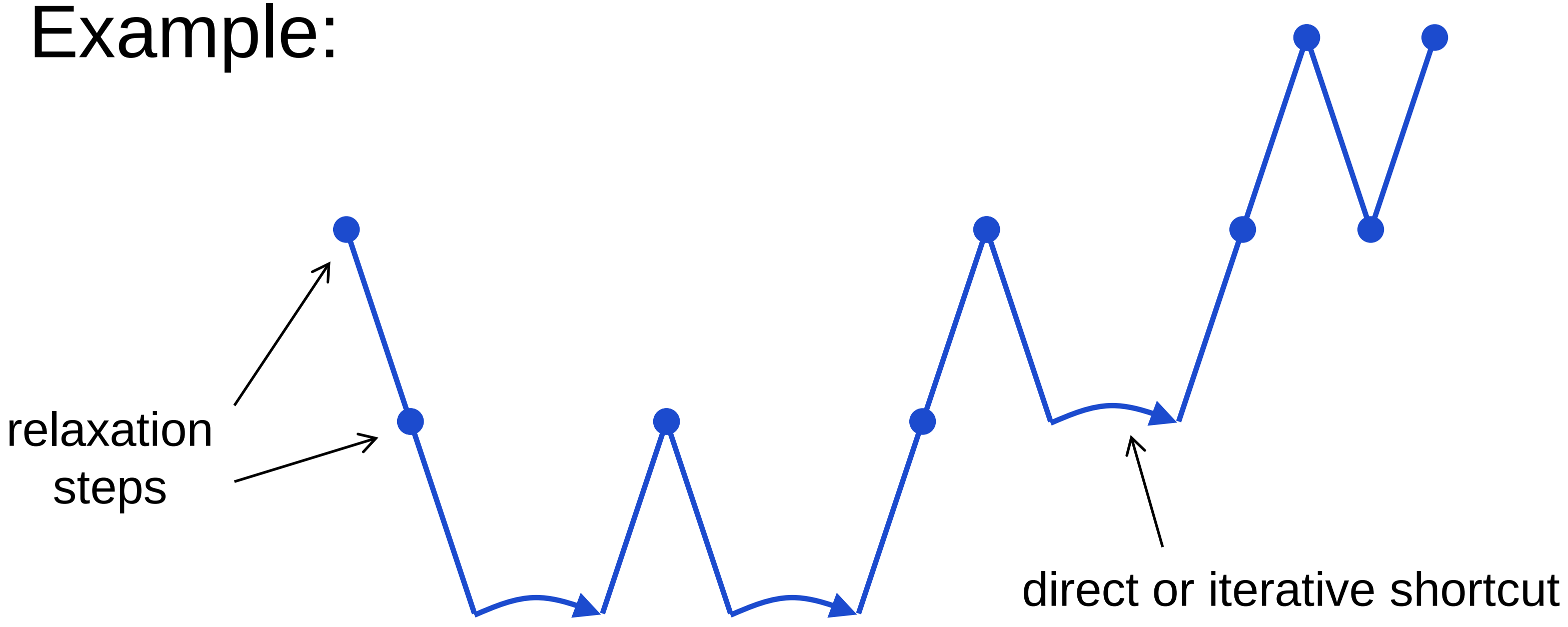
Full MG V-Cycle

How many iterations?

Standard Approaches

Multigrid Cycles

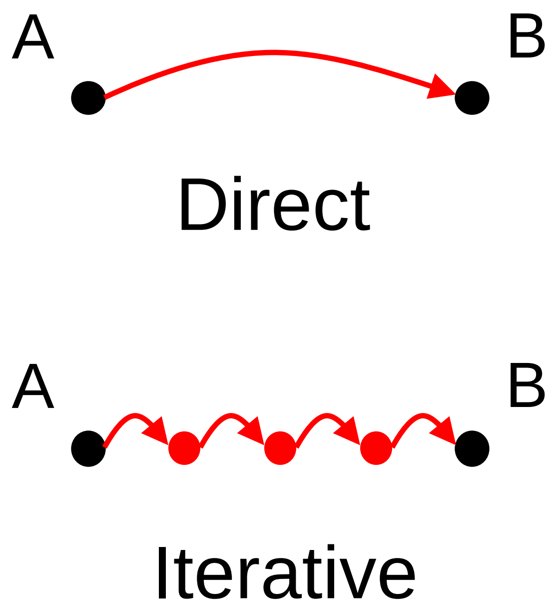
- Generalize the idea of what a multigrid cycle can look like
- Example:

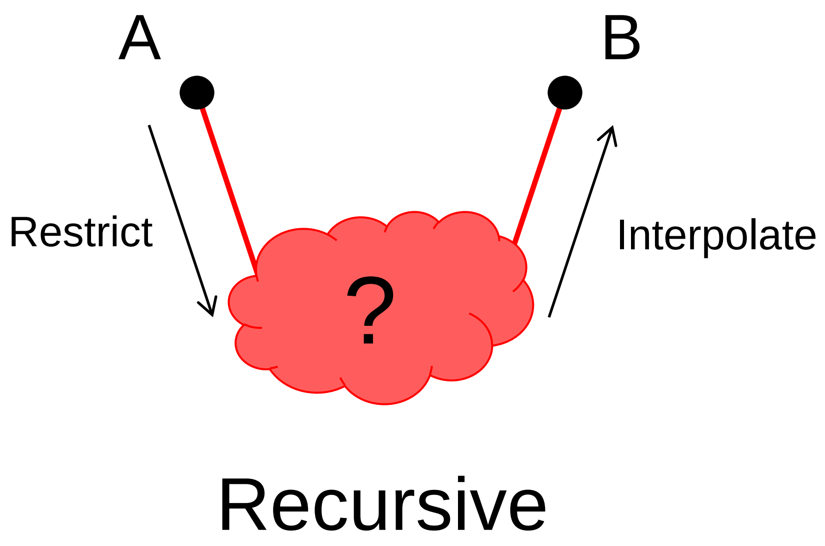


- Goal: Auto-tune cycle shape for specific usage

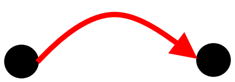
Algorithmic Choice in Multigrid

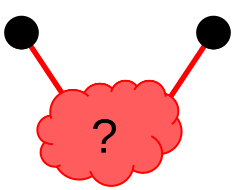
- Need framework to make fair comparisons
- Perspective of a specific grid resolution
- How to get from A to B?





Auto-tuning the V-cycle



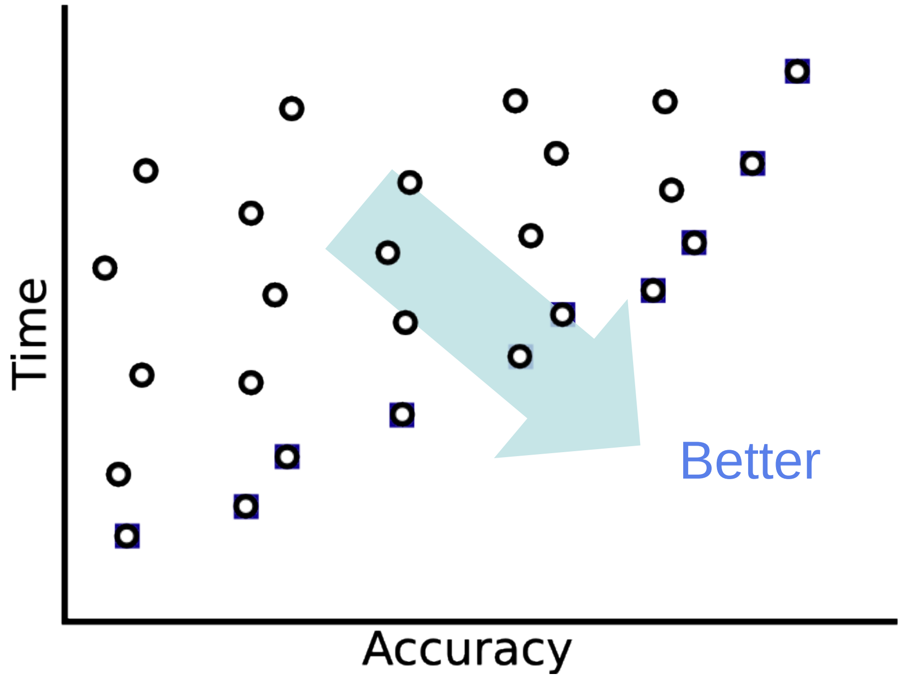



```
transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
{
    // Base case
    // Direct solve

        OR

    // Base case
    // Iterative solve at current resolution

        OR

    // Recursive case
    // For some number of iterations
        // Relax
        // Compute residual and restrict
        // Call Multigrid_i for some i
        // Interpolate and correct
        // Relax
}
```

- Algorithmic choice
  - Shortcut base cases
  - Recursively call some optimized sub-cycle
- Iterations and recursive accuracy let us explore accuracy versus performance space
- Only remember “best” versions

Optimal Subproblems

- Plot all cycle shapes for a given grid resolution:

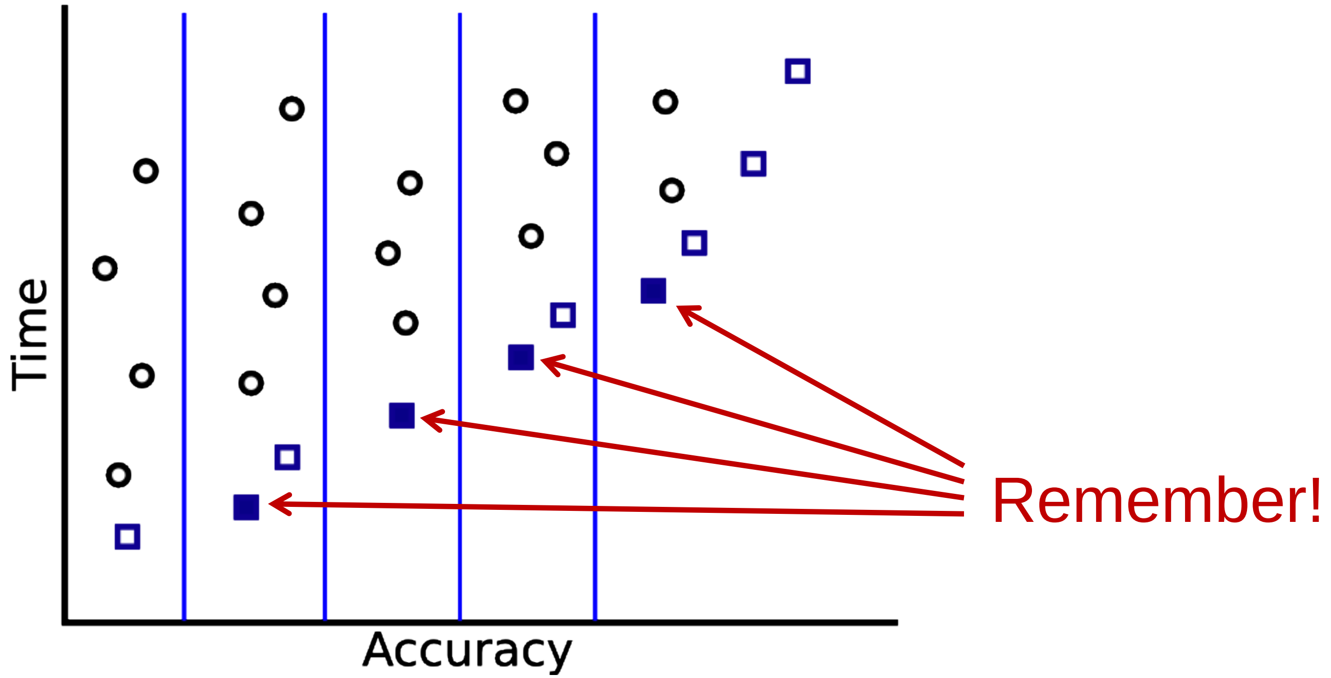


Keep only the optimal ones!

- Idea: Maintain a **family** of optimal algorithms for each grid resolution

The Discrete Solution

- Problem: Too many optimal cycle shapes to remember



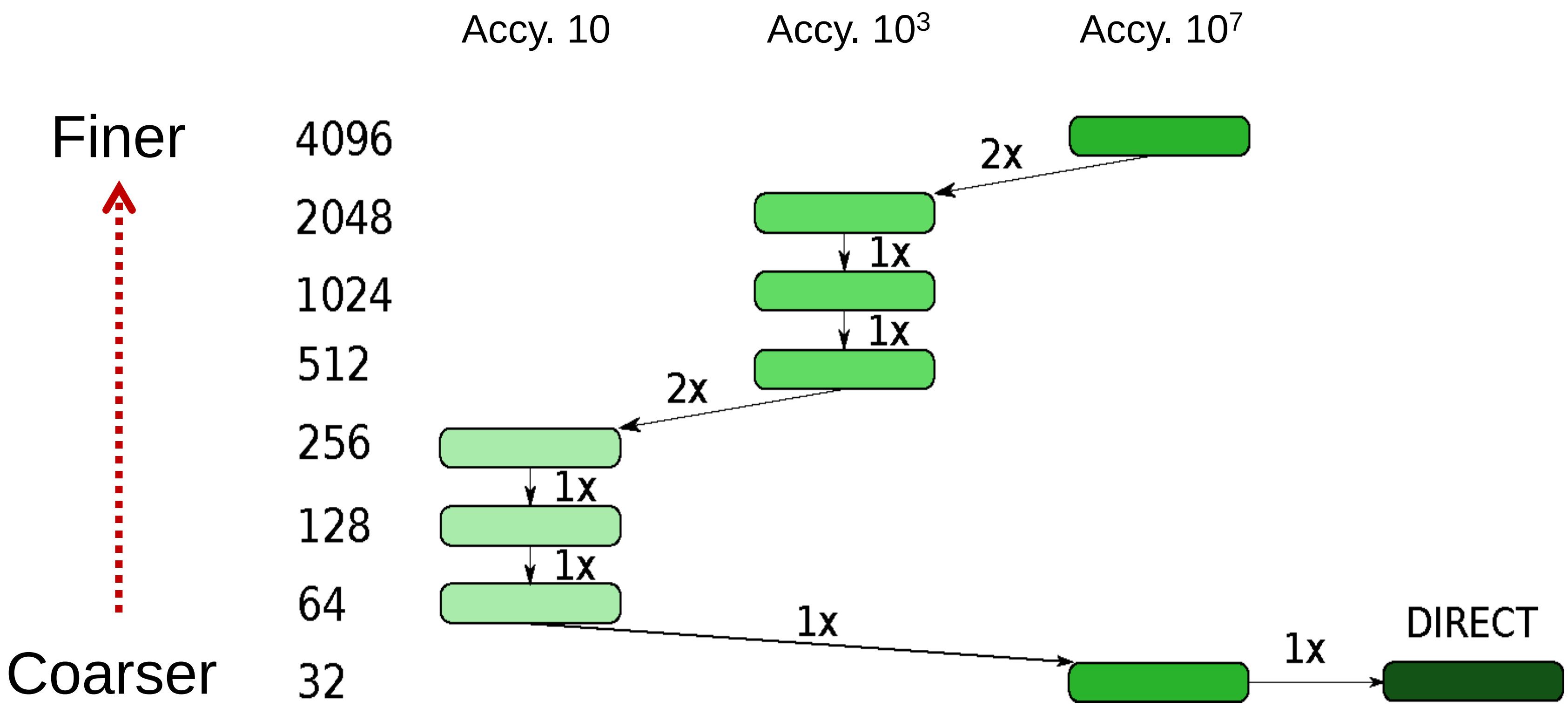
- Solution: Remember the fastest algorithms for a discrete set of accuracies

Use Dynamic Programming to Manage Auto-tuning Search

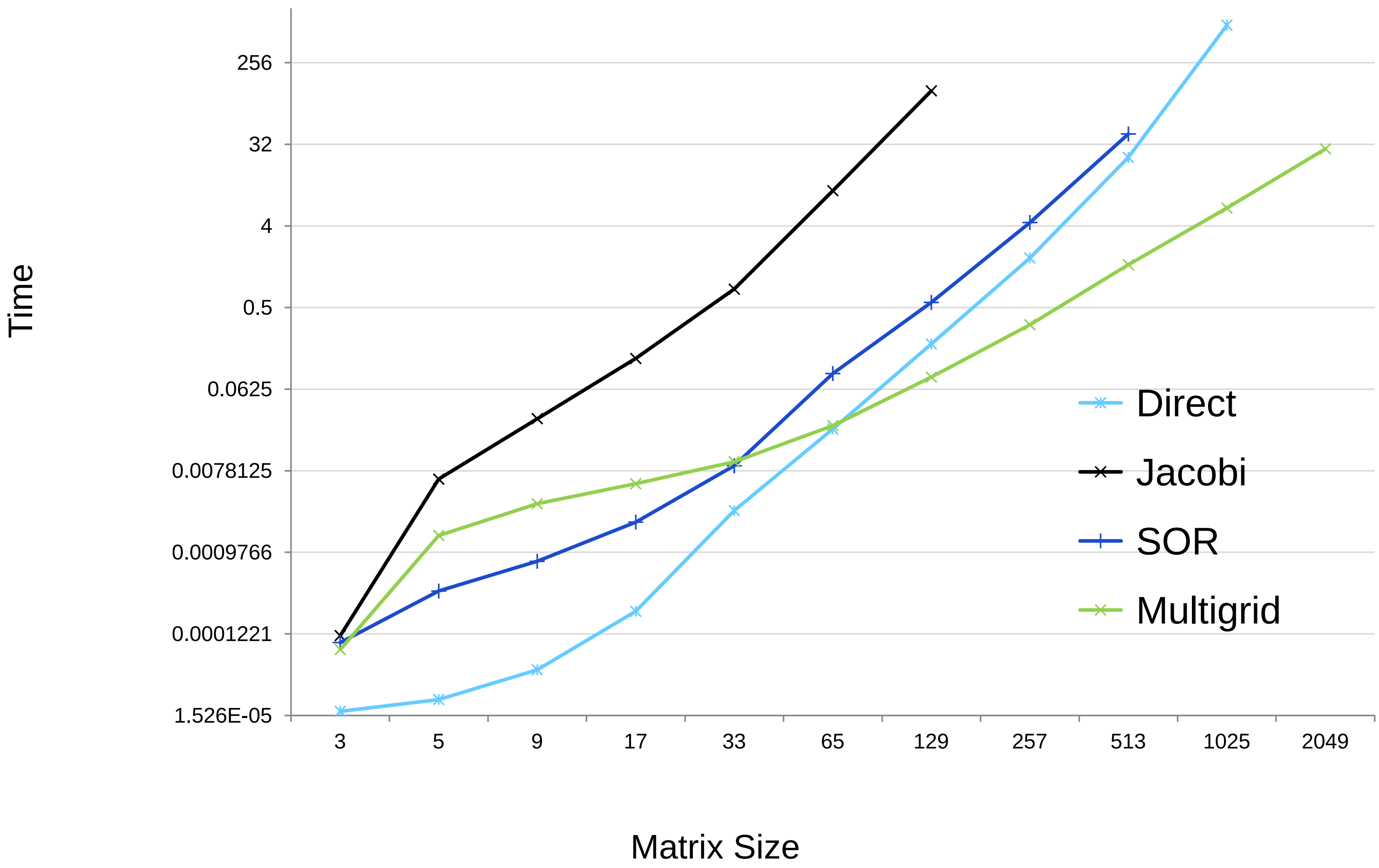
- Only search cycle shapes that utilize optimized sub-cycles in recursive calls
- Build optimized algorithms from the bottom up

- Allow shortcuts to stop recursion early
- Allow multiple iterations of sub-cycles to explore time vs. accuracy space

Example: Auto-tuned 2D Poisson's Equation Solver

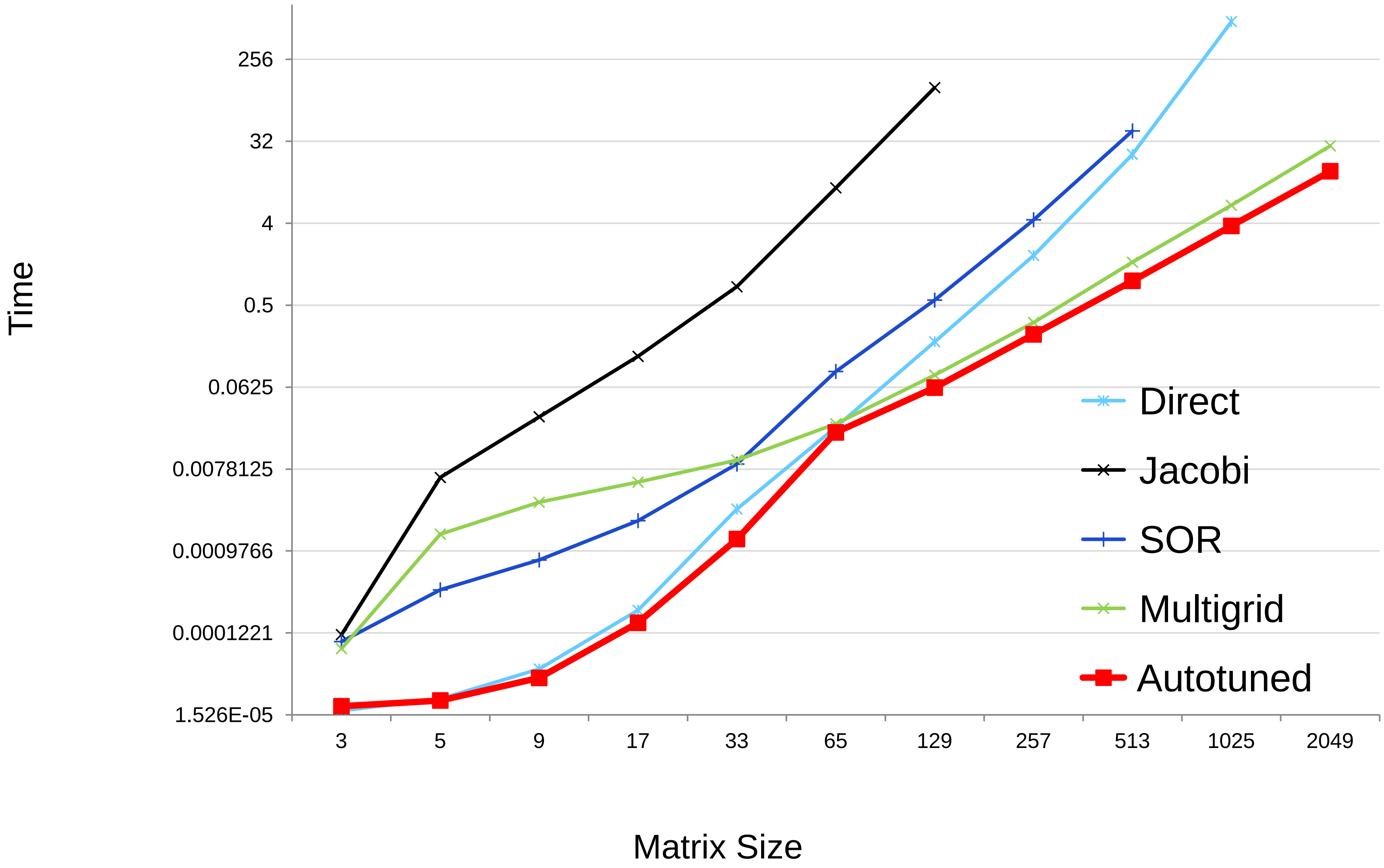


Poisson

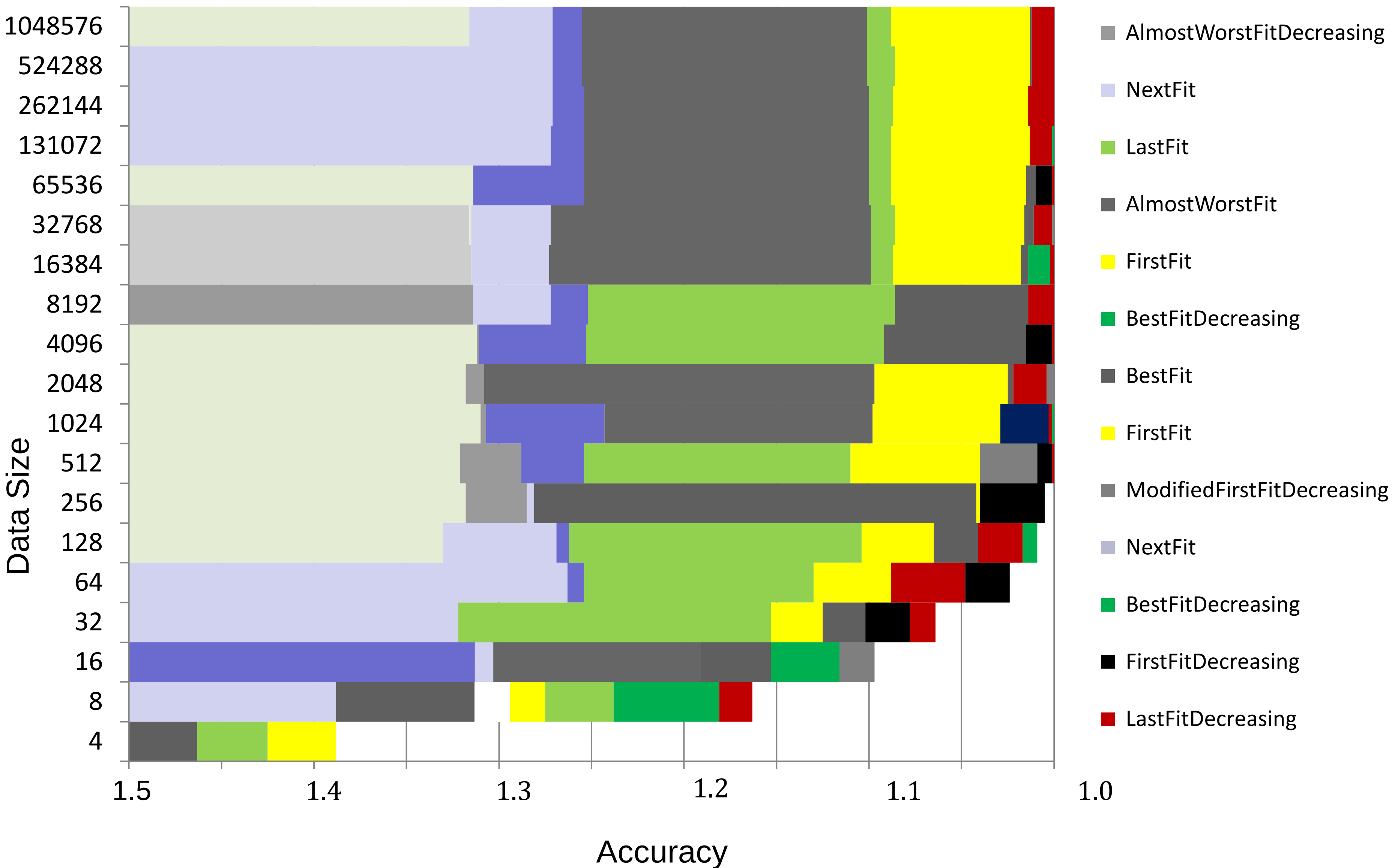


Time
256
32
4
0.5
0.0625
0.0078125
0.0009766
0.0001221
1.526E-05
3
5
9
17
33
65
129
257
513
1025
2049
Matrix Size
Direct
Jacobi
SOR
Multigrid

Poisson


Time
256
32
4
0.5
0.0625
0.0078125
0.0009766
0.0001221
1.526E-05
3
5
9
17
33
65
129
257
513
1025
2049
Matrix Size
Direct
Jacobi
SOR
Multigrid
Autotuned

Binpacking – Algorithmic Choices



Conclusion

- Time has come for languages based on autotuning
- Convergence of multiple forces
  - The Multicore Menace
  - Future proofing when machine models are changing
  - Use more muscle (compute cycles) than brain (human cycles)
- PetaBricks – We showed that it can be done!
- Will programmers accept this model?
  - A little more work now to save a lot later
  - Complexities in testing, verification and validation