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

CSAIL

Outline

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Today: The Happily Oblivious Average Joe Programmer

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

Today: The Happily Oblivious Average Joe Programmer

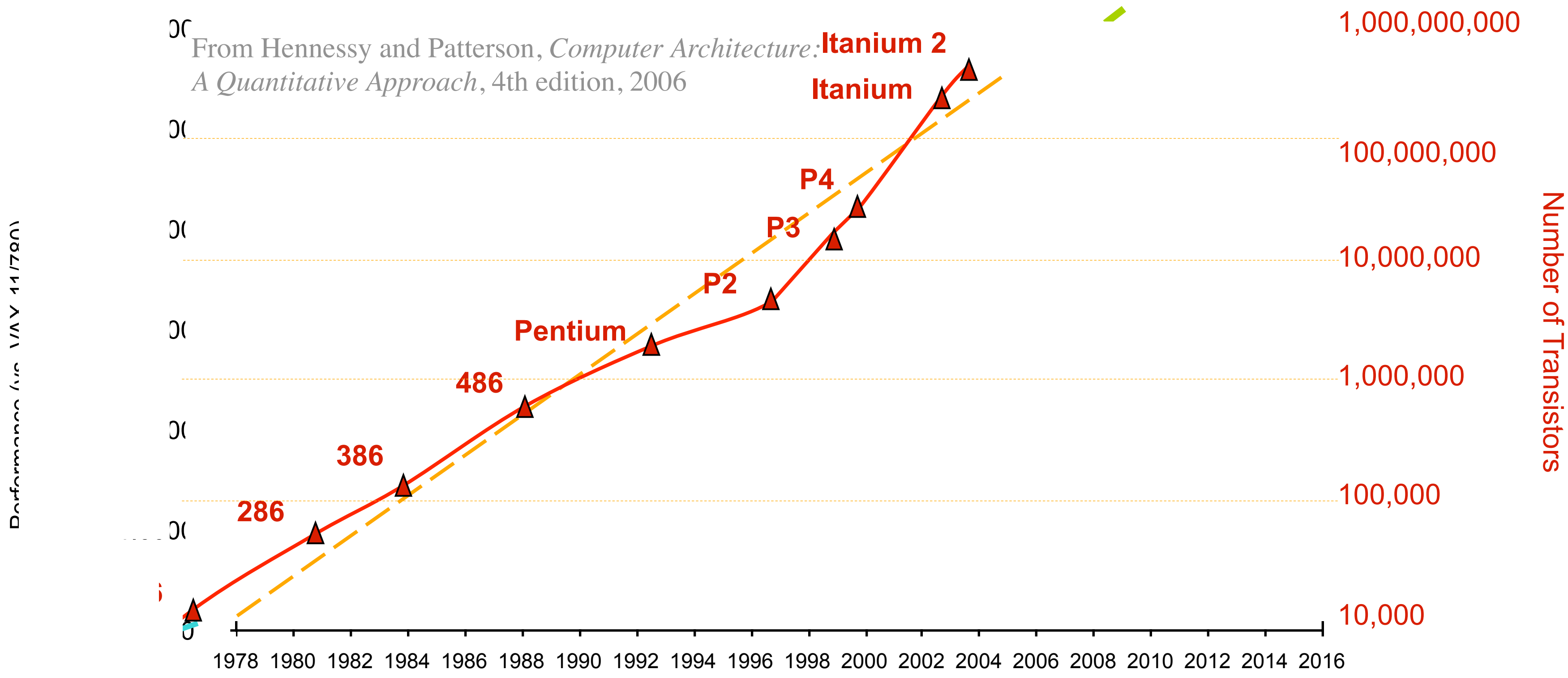
- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

- Parallel Programming is only practiced by a few experts

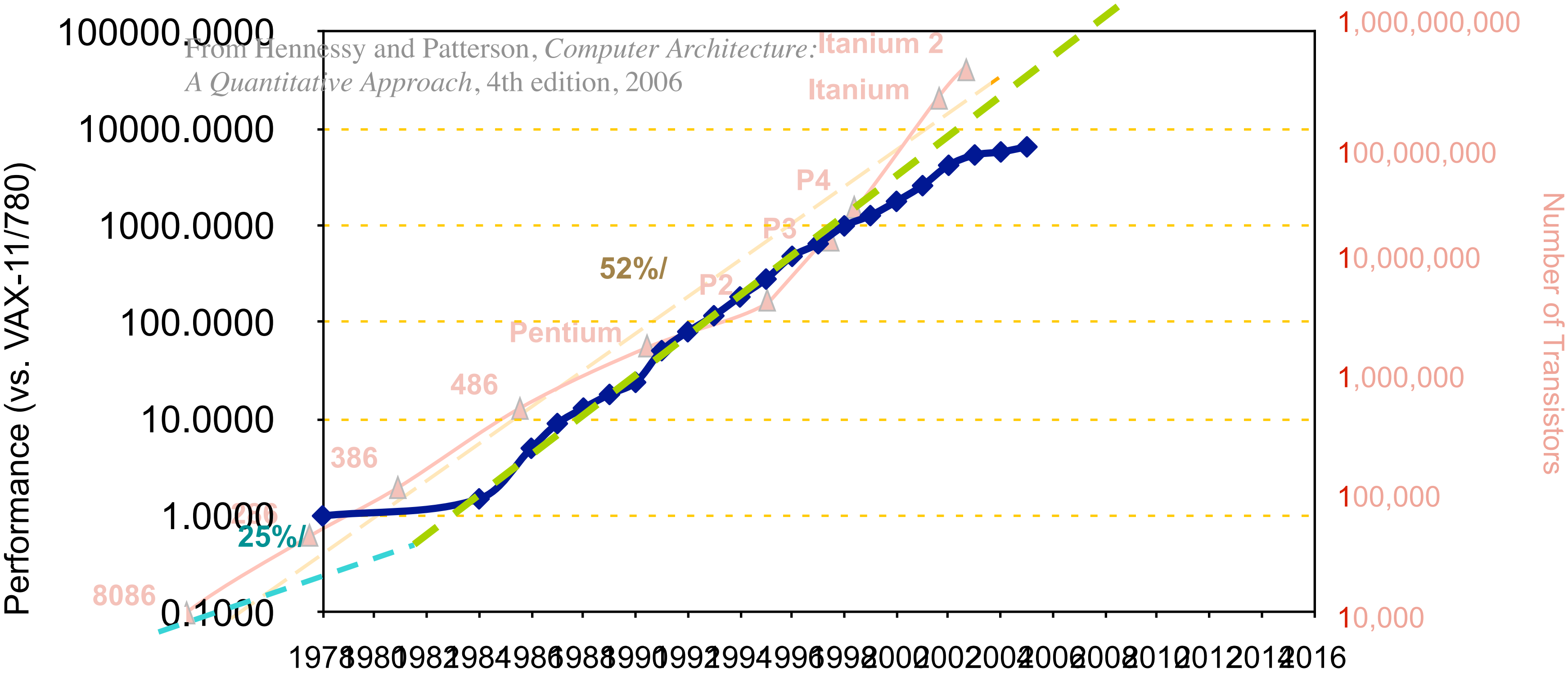
Moore’s Law



From David Patterson

Tuesday, October 25, 2011

Uniprocessor Performance (SPECint)



From David Patterson

Tuesday, October 25, 2011

Squandering of the Moore's Dividend

- 10,000x performance gain in 30 years! (~46% per year)
- Where did this performance go?

Squandering of the Moore's Dividend

- 10,000x performance gain in 30 years! (~46% per year)
- Where did this performance go?
- Last decade we concentrated on correctness and programmer productivity

Squandering of the Moore's Dividend

- 10,000x performance gain in 30 years! (~46% per year)
- Where did this performance go?
- Last decade we concentrated on correctness and programmer productivity
- Little to no emphasis on performance

Squandering of the Moore's Dividend

- 10,000x performance gain in 30 years! (~46% per year)
- Where did this performance go?
- Last decade we concentrated on correctness and programmer productivity
- Little to no emphasis on performance
- This is reflected in:
  - Languages
  - Tools
  - Research
  - Education

Squandering of the Moore's Dividend

- 10,000x performance gain in 30 years! (~46% per year)
- Where did this performance go?
- Last decade we concentrated on correctness and programmer productivity
- Little to no emphasis on performance
- This is reflected in:
  - Languages
  - Tools
  - Research
  - Education
- Software Engineering: Only engineering discipline where performance or efficiency is not a central theme

Matrix Multiply

	Immutable
ms	17,094,152

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching

Matrix Multiply

	Immutable
ms	17,094,152

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

Matrix Multiply

	Immutable	Mutable
ms	17,094,152	77,826

219.7x

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

220x

Matrix Multiply

	Immutable	Mutable	Double Only
ms	17,094,152	77,826	32,800

219.7x

2.4x

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

522x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects
ms	17,094,152	77,826	32,800	15,306

219.7x (Immutable → Mutable)

2.4x (Mutable → Double Only)

2.2x (Double Only → No Objects)

- Abstraction and Software Engineering
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

1,117x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects
ms	17,094,152	77,826	32,800	15,306
	219.7x		2.2x	
		2.4x		

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

1,117x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C
ms	17,094,152	77,826	32,800	15,306	7,530
	219.7x			2.2x	
			2.4x		2.1x

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

2,271x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C	Transposed
ms	17,094,152	77,826	32,800	15,306	7,530	2,275
	219.7x		2.2x		3.4x	
		2.4x		2.1x		

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

7,514x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388
	219.7x		2.2x		3.4x		
		2.4x		2.1x		1.7x	

12,316x

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled	Vectorized
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388	511
	219.7x		2.2x		3.4x		2.8x	
		2.4x		2.1x		1.7x		

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

33,453x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled	Vectorized	BLAS MxM
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388	511	196
	219.7x		2.2x		3.4x		2.8x		
		2.4x		2.1x		1.7x		2.7x	

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

87,042x

Matrix Multiply

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled	Vectorized	BLAS MxM	BLAS Parallel
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388	511	196	58
	219.7x		2.2x		3.4x		2.8x		3.5x	
		2.4x		2.1x		1.7x		2.7x		

- ~~Abstraction and Software Engineering~~
  - Immutable Types
  - Dynamic Dispatch
  - Object Oriented
- High Level Languages
- Memory Management
  - Transpose for unit stride
  - Tile for cache locality
- Vectorization
- Prefetching
- Parallelization

296,260x

Matrix Multiply

- Typical Software Engineering Approach
  - In Java
  - Object oriented
  - Immutable
  - Abstract types
  - No memory optimizations
  - No parallelization

296,260x

- Good Performance Engineering Approach
  - In C/Assembly
  - Memory optimized (blocked)
  - BLAS libraries
  - Parallelized (to 4 cores)

Matrix Multiply

- Typical Software Engineering Approach
  - In Java
  - Object oriented
  - Immutable
  - Abstract types
  - No memory optimizations
  - No parallelization

296,260x

- Good Performance Engineering Approach

  In C/Assembly

  Memory optimized (blocked)

  BLAS libraries

  Parallelized (to 4 cores)

- In Comparison: Lowest to Highest MPG in transportation

14,700x

Matrix Multiply

- Typical Software Engineering Approach
  - In Java
  - Object oriented
  - Immutable
  - Abstract types
  - No memory optimizations
  - No parallelization

296,260x

- Good Performance Engineering Approach

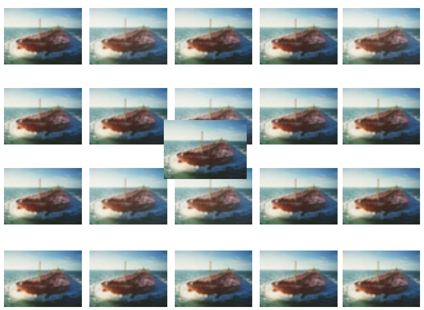
  In C/Assembly

  Memory optimized (blocked)

  BLAS libraries

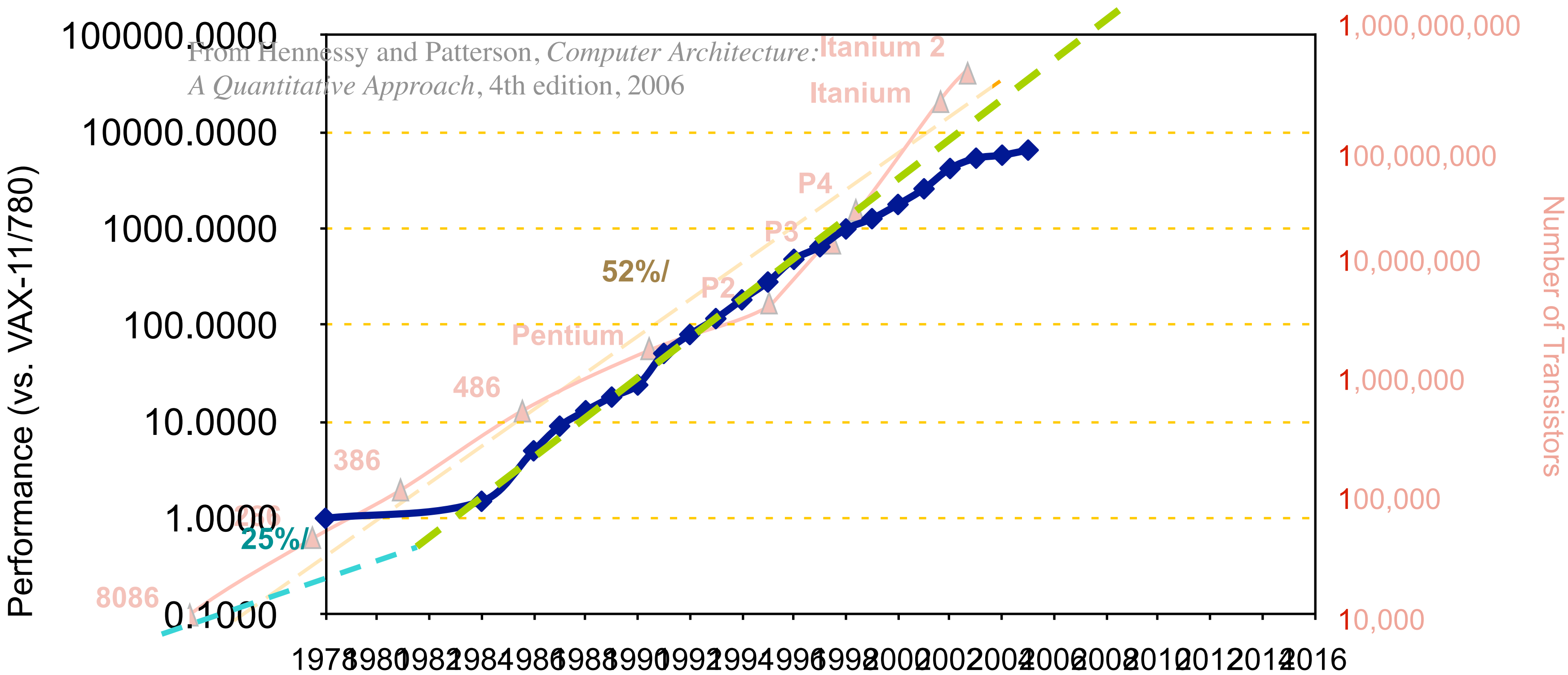
  Parallelized (to 4 cores)

- In Comparison: Lowest to Highest MPG in transportation



294,000x

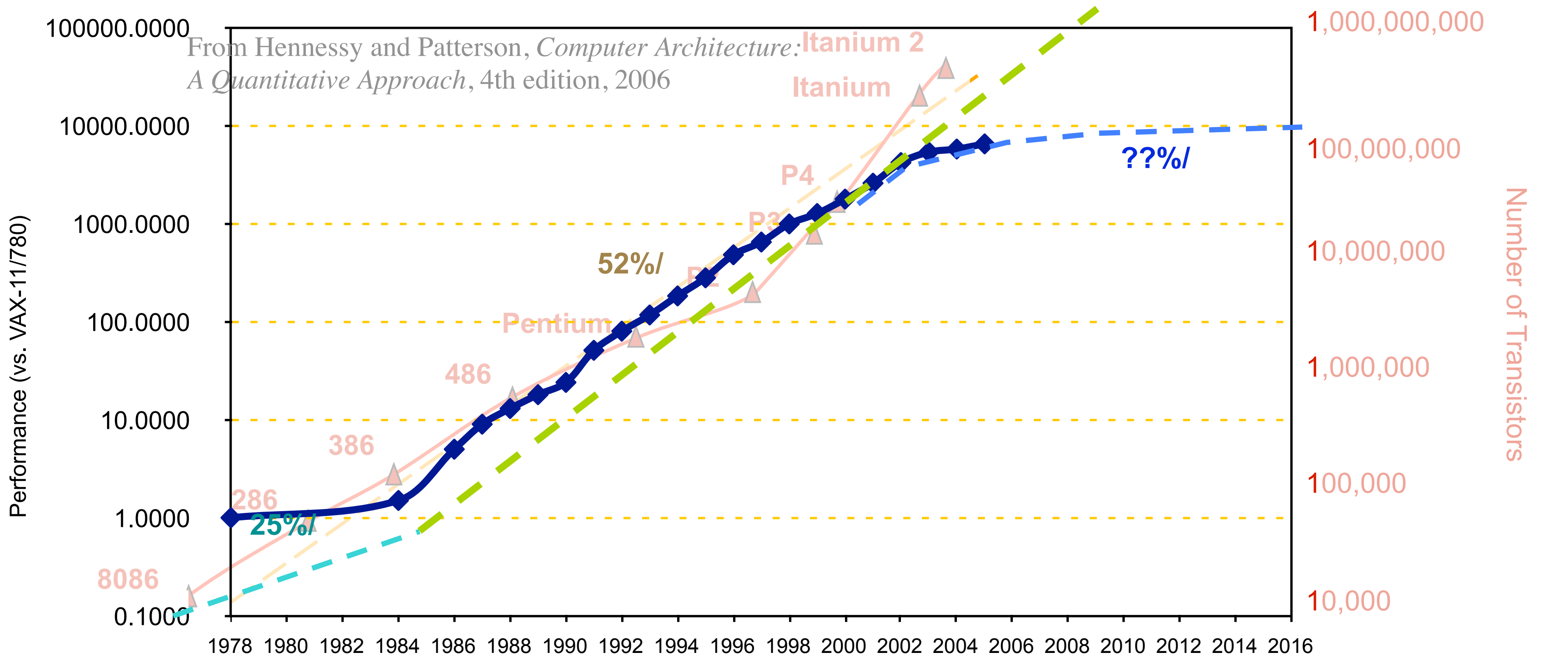
Uniprocessor Performance (SPECint)



From David Patterson

Tuesday, October 25, 2011

Uniprocessor Performance (SPECint)



From David Patterson

Tuesday, October 25, 2011

Performance and Parallelism

- No more automatic performance gains

  → Performance has to come from somewhere else

  - Better languages
  - Disciplined programming
  - Performance engineering
  - Plus…

Performance and Parallelism

- No more automatic performance gains

  →Performance has to come from somewhere else

  - Better languages
  - Disciplined programming
  - Performance engineering
  - Plus…
- Parallelism
  - Moore's low morphed from providing performance to providing parallelism
  - But…Parallelism IS performance

Joe the Parallel Programmer

- Moore's law is not bringing anymore performance gains
- If Joe needs performance he has to deal with multicores
  - Joe has to deal with performance
  - Joe has to deal with parallelism

Can Joe Handle This?

Today

Programmer is oblivious to performance.

Can Joe Handle This?

Current Trajectory

Programmer handles parallelism and performance turning

Today

Programmer is oblivious to performance.

Can Joe Handle This?

Today

Programmer is oblivious to performance.

Current Trajectory

Programmer handles parallelism and performance turning

Better Trajectory

Programmer handles concurrency. Compiler finds best parallel mapping and optimize for performance

Conquering the Multicore Menace

Conquering the Multicore Menace

- Parallelism Extraction
  - The world is parallel,
    but most computer science is based in sequential thinking
  - Parallel Languages
    - Natural way to describe the maximal concurrency in the problem
  - Parallel Thinking
    - Theory, Algorithms, Data Structures → Education

Conquering the Multicore Menace

- Parallelism Extraction
  - The world is parallel,
    but most computer science is based in sequential thinking
  - Parallel Languages
    - Natural way to describe the maximal concurrency in the problem
  - Parallel Thinking
    - Theory, Algorithms, Data Structures → Education
- Parallelism Management
  - Mapping algorithmic parallelism to a given architecture
  - Find the best performance possible

Outline

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

In the mean time…….the experts practicing

- They needed to get the last ounce of the performance from hardware
- They had problems that are too big or too hard
- They worked on the biggest newest machines
- Porting the software to take advantage of the latest hardware features
- Spending years (lifetimes) on a specific kernel

Lifetime of Software >> Hardware

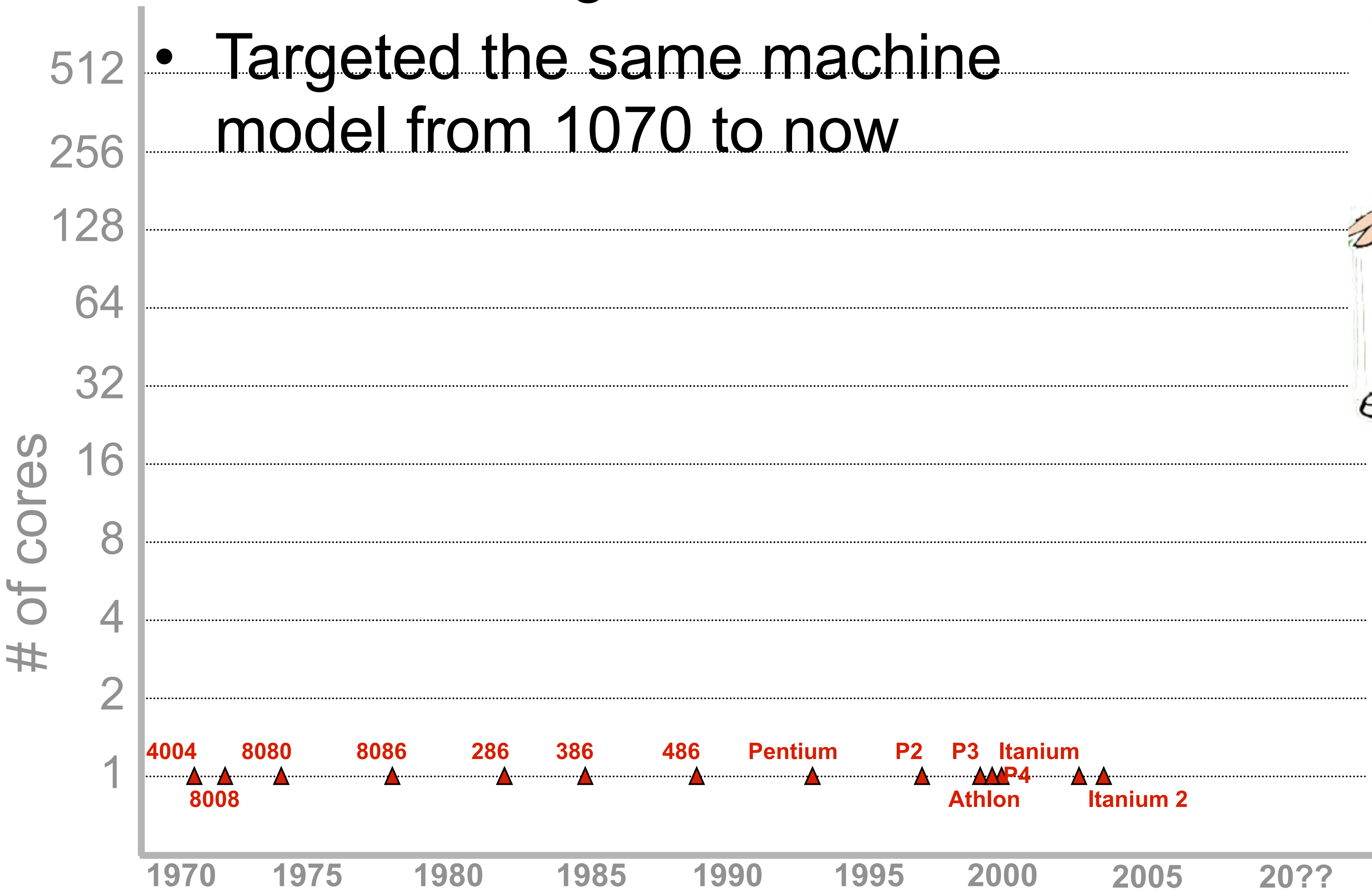
- Lifetime of a software application is 30+ years

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

- Multiple Ports
- “Software Quality deteriorates in each port
- Huge problem for these expert programmers

Not a problem for Joe

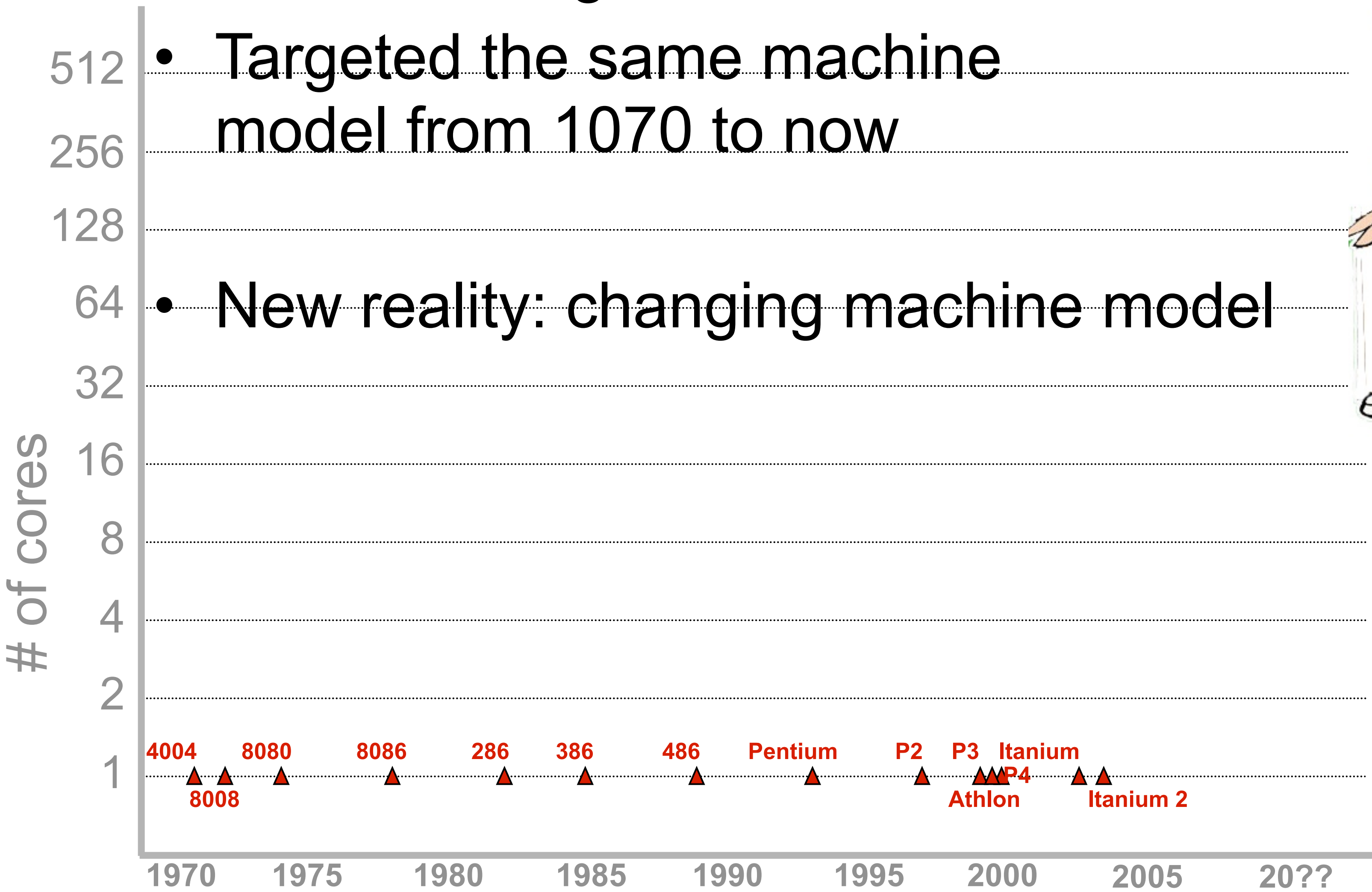
- Moore's law gains were sufficient
- Targeted the same machine model from 1070 to now



Not a problem for Joe

- Moore's law gains were sufficient
- Targeted the same machine model from 1070 to now

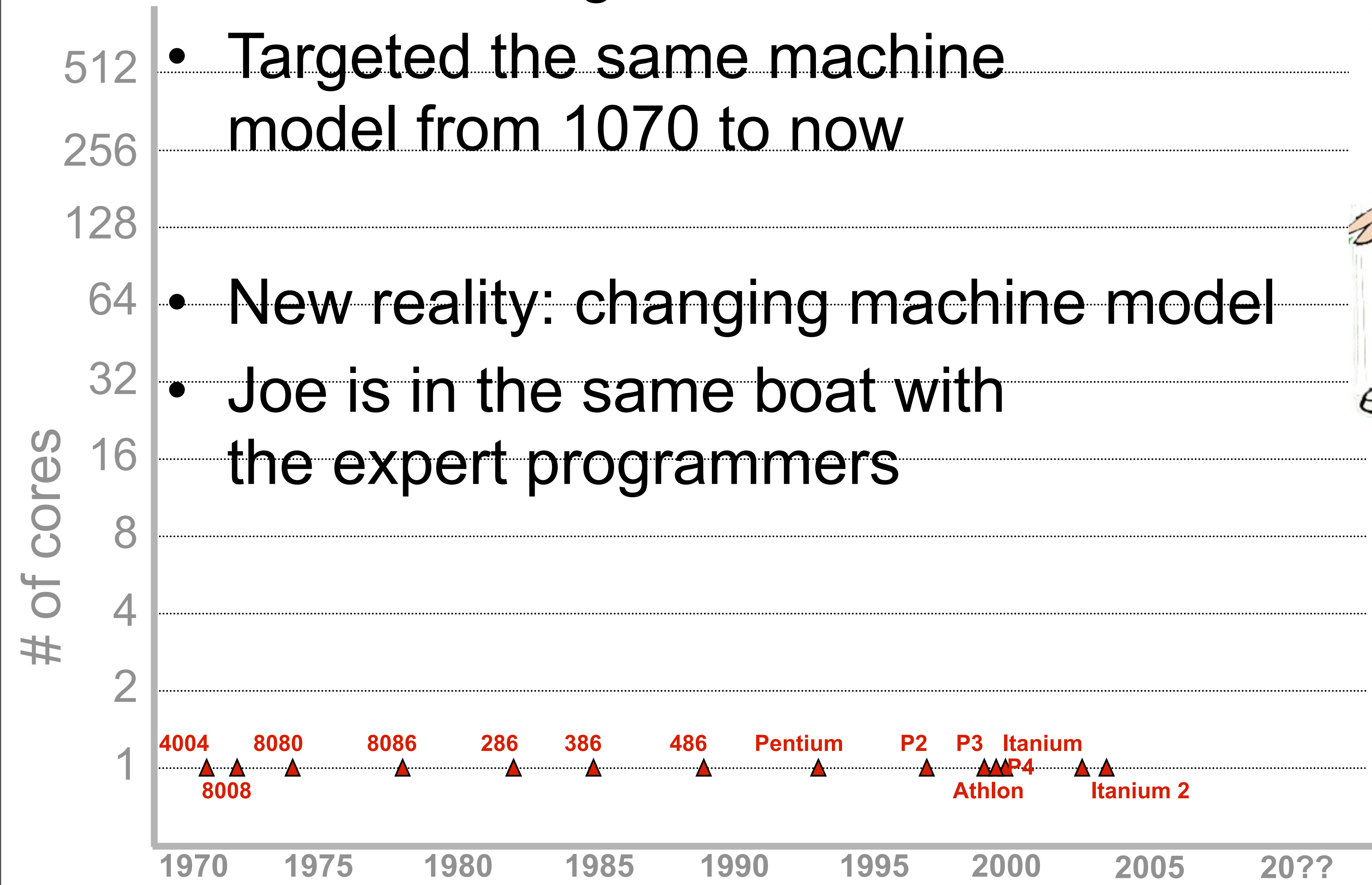
- New reality: changing machine model



Not a problem for Joe

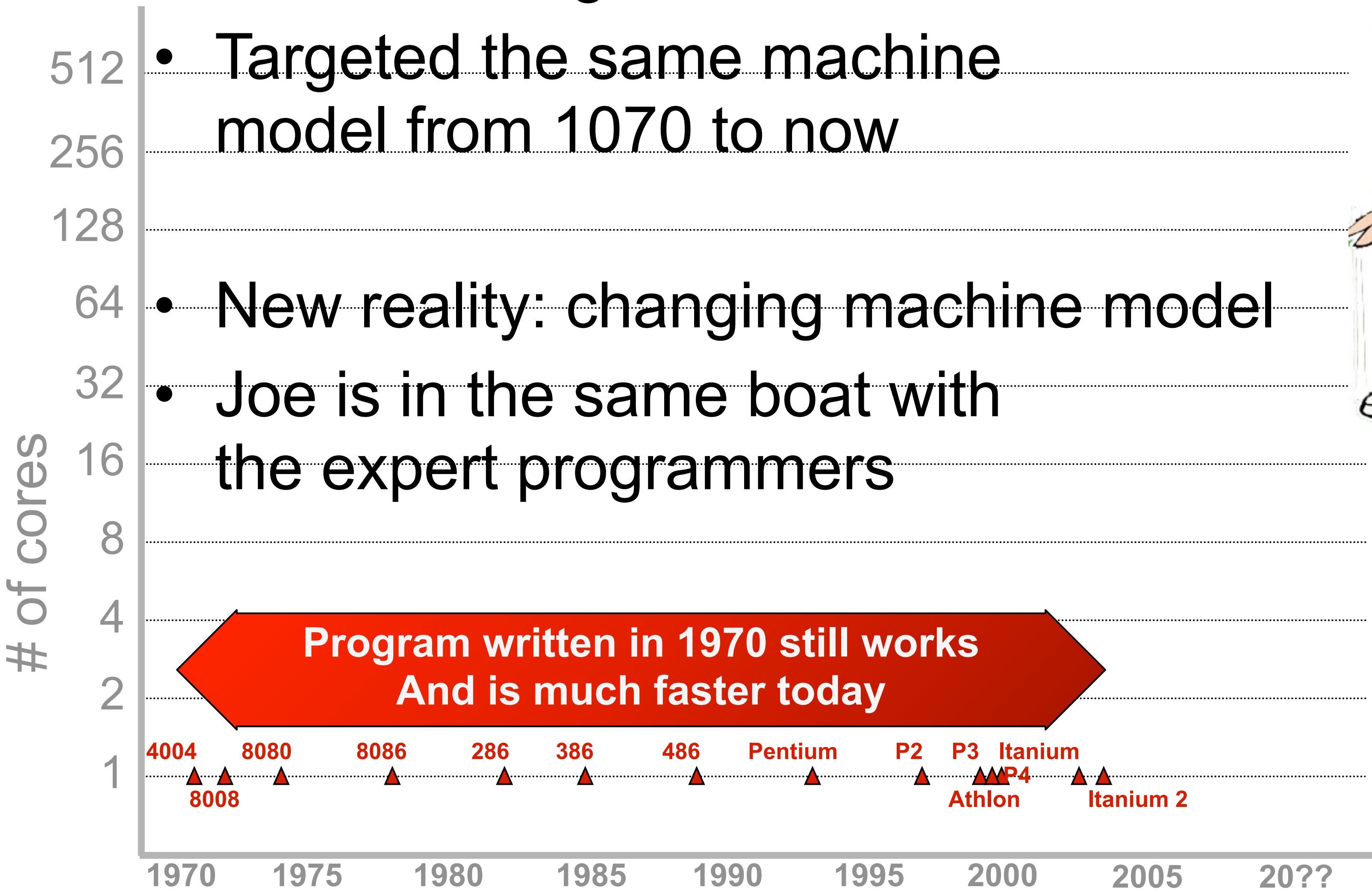
- Moore’s law gains were sufficient
- Targeted the same machine model from 1070 to now

- New reality: changing machine model
- Joe is in the same boat with the expert programmers



Not a problem for Joe

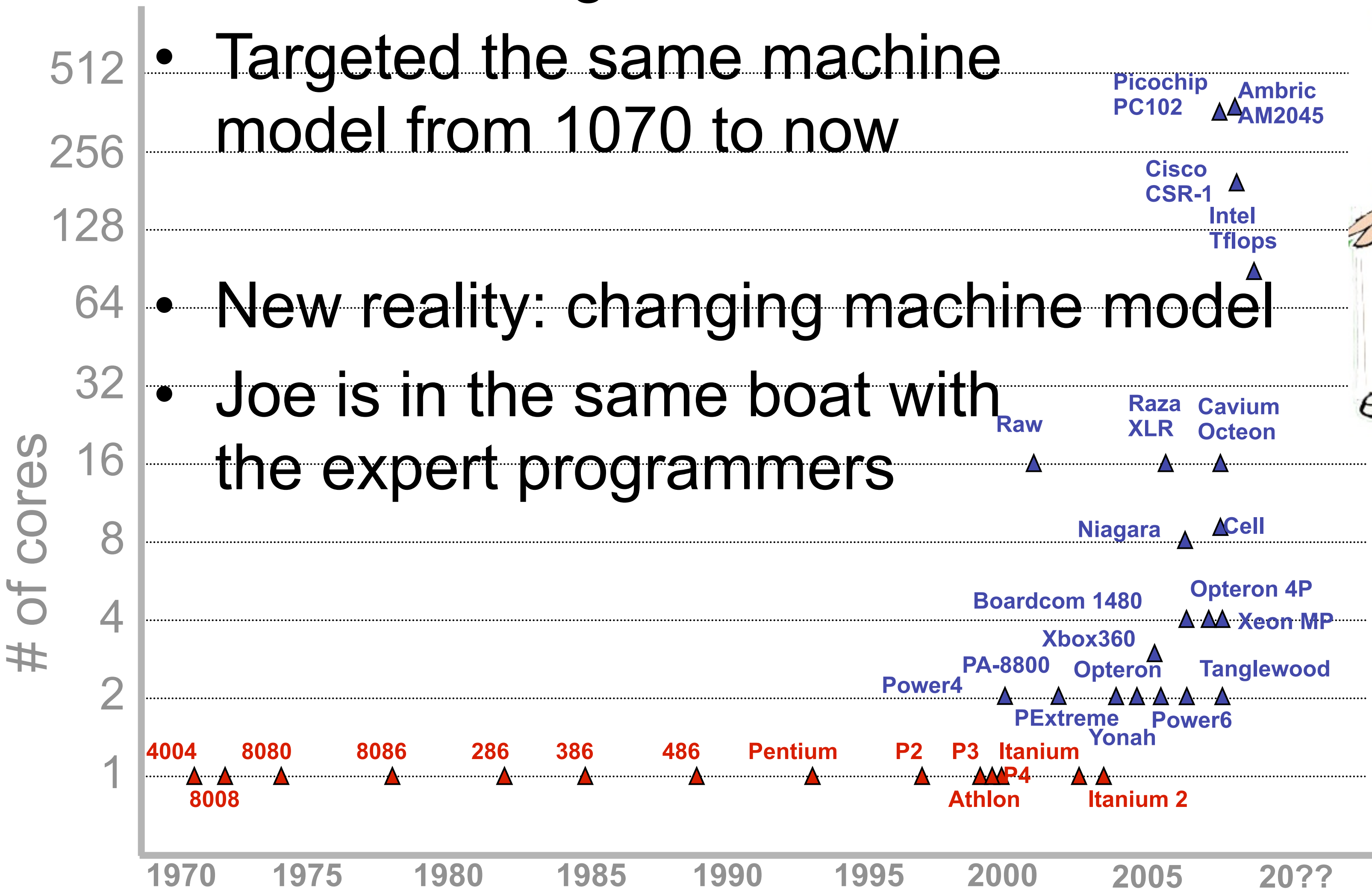
- Moore's law gains were sufficient
- Targeted the same machine model from 1070 to now
- New reality: changing machine model
- Joe is in the same boat with the expert programmers



Not a problem for Joe

- Moore's law gains were sufficient
- Targeted the same machine model from 1070 to now

- New reality: changing machine model
- Joe is in the same boat with the expert programmers



Tuesday, October 25, 2011

Future Proofing Software

- No single machine model anymore
  - Between different processor types
  - Between different generation within the same family
- Programs need to be written-once and use anywhere, anytime
  - Java did it for portability
  - We need to do it for performance

Languages and Future Proofing

- To be an effective language that can future-proof programs
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user
  - Features that are automatable and variable → hide from the user

Languages and Future Proofing

- **A lot now**
  - Expose the architectural details
  - Good performance now
  - In a local minima
  - Will be obsolete soon
  - Heroic effort needed to get out
  - Ex: MPI

- To be an effective language that can future-proof programs
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user
  - Features that are automatable and variable → hide from the user

Languages and Future Proofing

- **A lot now**
  - Expose the architectural details
  - Good performance now
  - In a local minima
  - Will be obsolete soon
  - Heroic effort needed to get out
  - Ex: MPI

- **A little forever**
  - Hide the architectural details
  - Good solutions not visible
  - Mediocre performance
  - But will work forever
  - Ex: HPF

- To be an effective language that can future-proof programs
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user
  - Features that are automatable and variable → hide from the user

Languages and Future Proofing

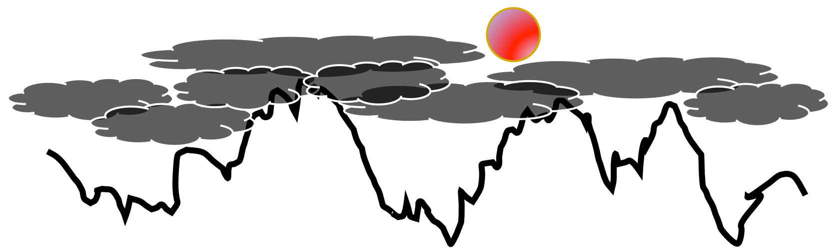
- **A lot now**
  - Expose the architectural details
  - Good performance now
  - In a local minima
  - Will be obsolete soon
  - Heroic effort needed to get out
  - Ex: MPI
- **A little forever**
  - Hide the architectural details
  - Good solutions not visible
  - Mediocre performance
  - But will work forever
  - Ex: HPF

- To be an effective language that can future-proof programs
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user

Languages and Future Proofing

- **A lot now**
  - Expose the architectural details
  - Good performance now
  - In a local minima
  - Will be obsolete soon
  - Heroic effort needed to get out
  - Ex: MPI

- **A little forever**
  - Hide the architectural details
  - Good solutions not visible
  - Mediocre performance
  - But will work forever
  - Ex: HPF

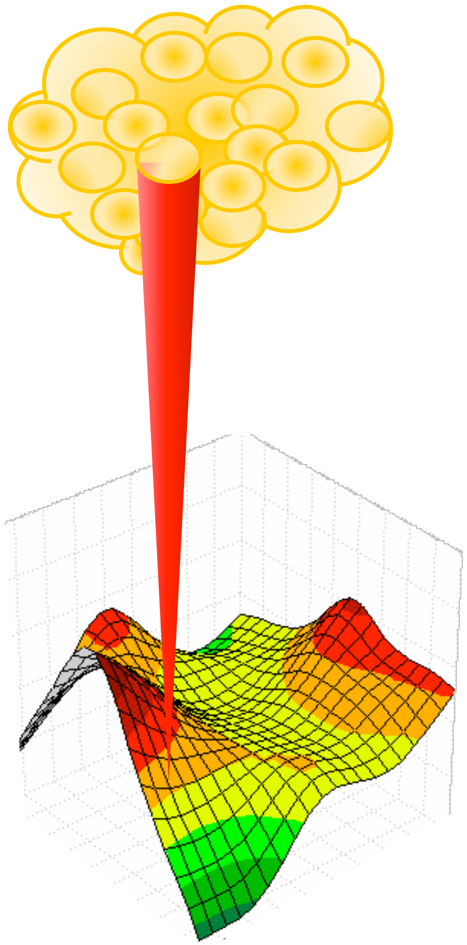


- To be an effective language that can future-proof programs
  - Restrict the choices when a property is hard to automate or constant across architectures of current and future → expose to the user
  - Features that are automatable and variable → hide from the user

Outline

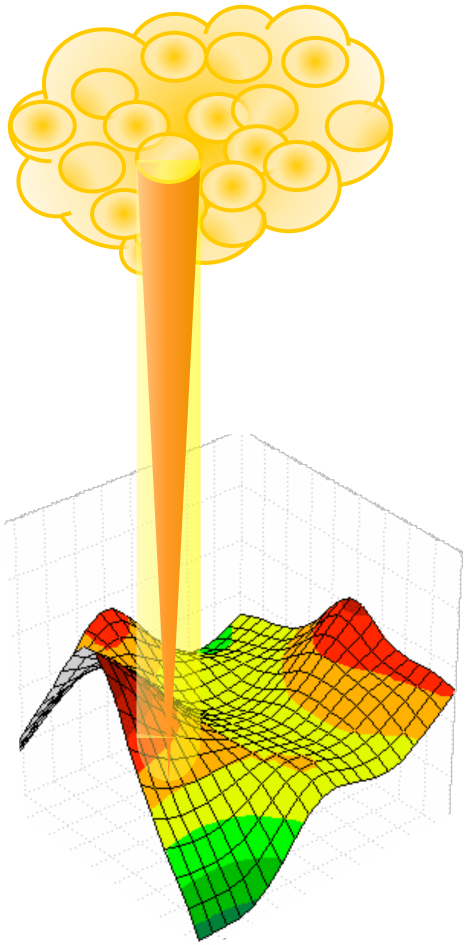
- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Ancient Days…



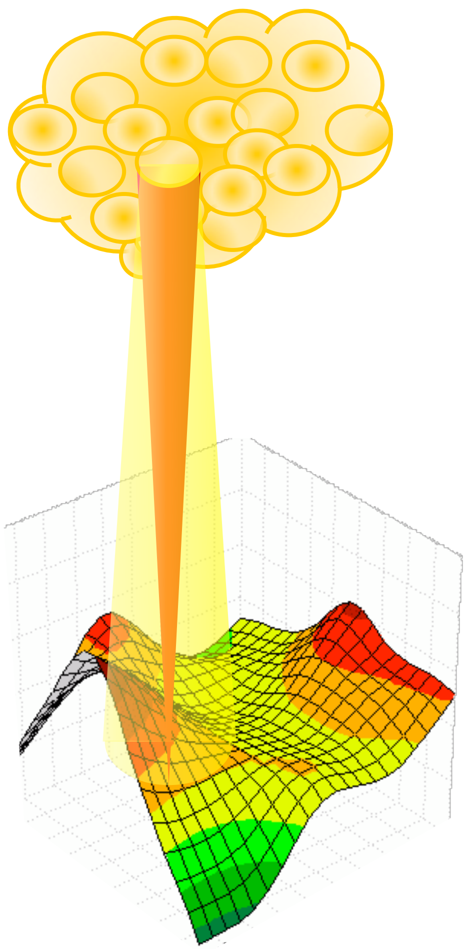
- Computers had limited power
- Compiling was a daunting task
- Languages helped by limiting choice
- Overconstraint programming languages that express only a single choice of:
  - Algorithm
  - Iteration order
  - Data layout
  - Parallelism strategy

…as we progressed….



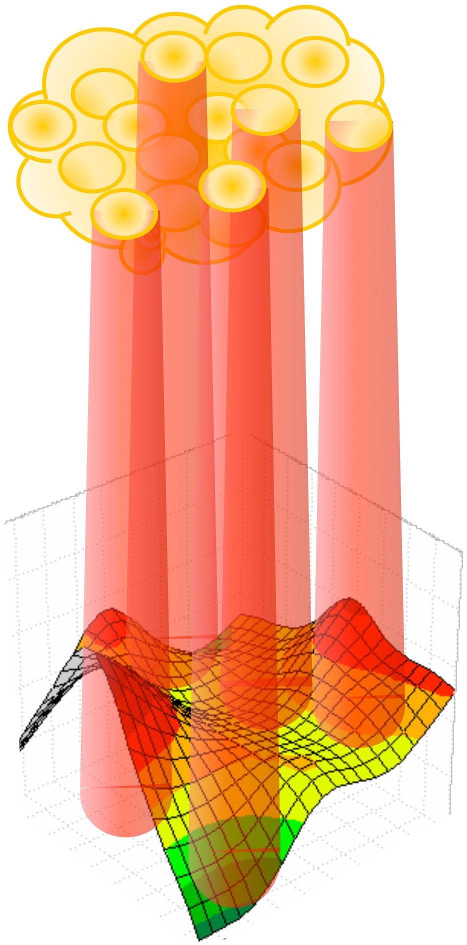
- Computers got faster
- More cycles available to the compiler
- Wanted to optimize the programs, to make them run better and faster

…and we ended up at

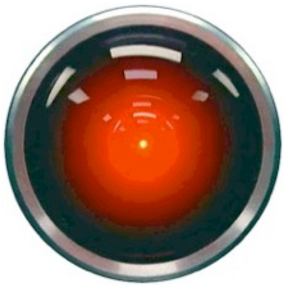


- Computers are extremely powerful
- Compilers want to do a lot
- But…the same old overconstraint languages
  – They don't provide too many choices
- Heroic analysis to rediscover some of the choices
  – Data dependence analysis
  – Data flow analysis
  – Alias analysis
  – Shape analysis
  – Interprocedural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

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Observation 1: Algorithmic Choice

Observation 1: 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.)

Observation 1: 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.

Observation 1: 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 2: Natural Parallelism

Observation 2: Natural Parallelism

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

Observation 2: Natural Parallelism

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

Observation 2: 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))

Observation 2: 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

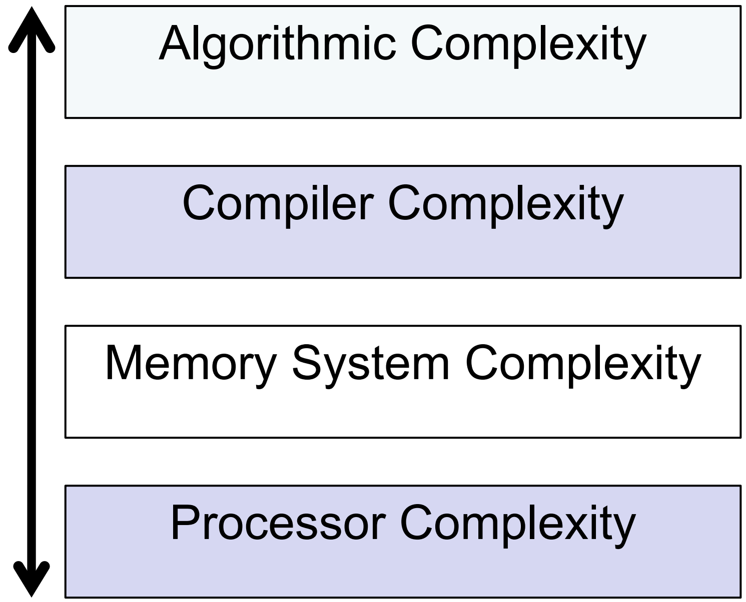
Observation 3: Autotuning

Observation 3: Autotuning

- Good old days → model based optimization

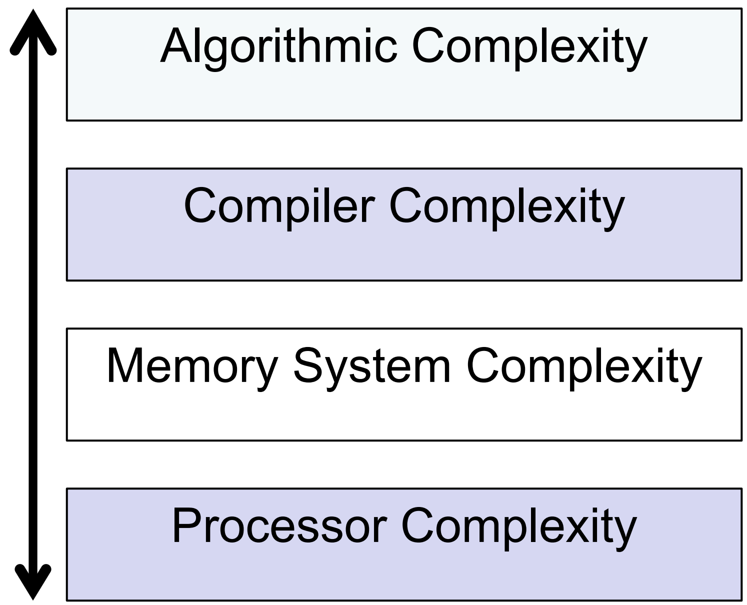
Observation 3: Autotuning

- 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



Observation 3: 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

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

PetaBricks Language

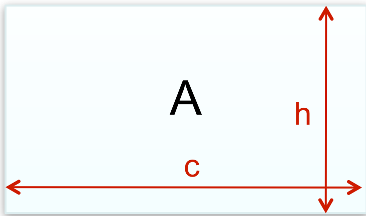
```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
   // Base case, compute a single element
   to(AB.cell(x,y) out)
   from(A.row(y) a, B.column(x) b) {
      out = dot(a, b);
  }
}
```

- 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);
  }
}
```

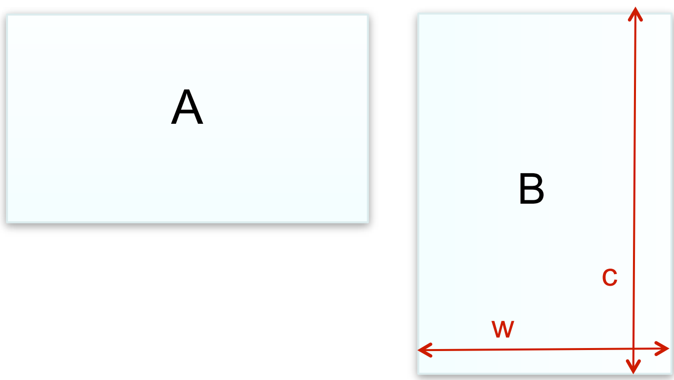
- Implicitly parallel description



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);
  }
}
```

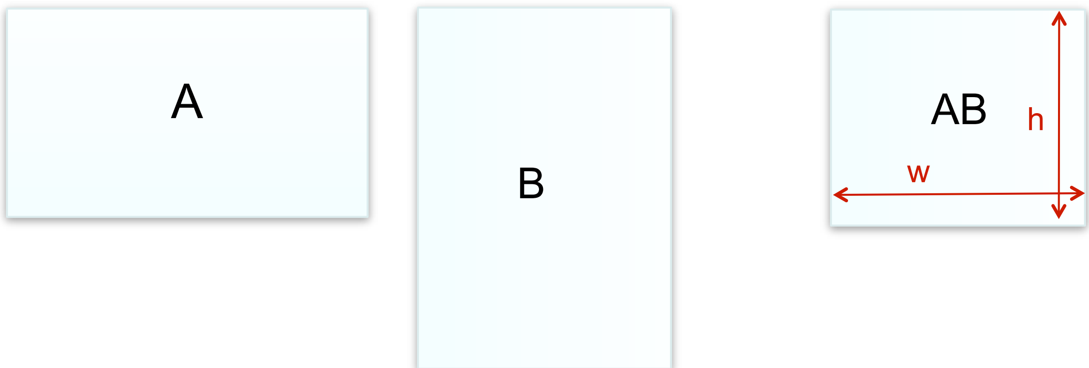
- Implicitly parallel description



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);
  }
}
```

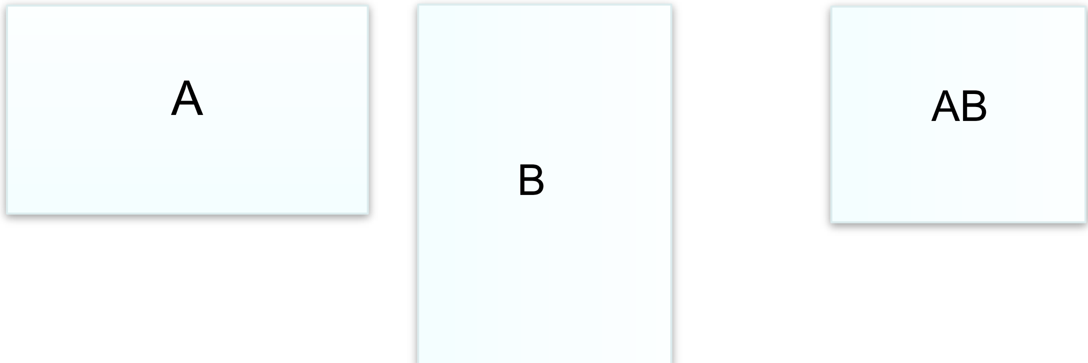
- Implicitly parallel description



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);
   }
}
```

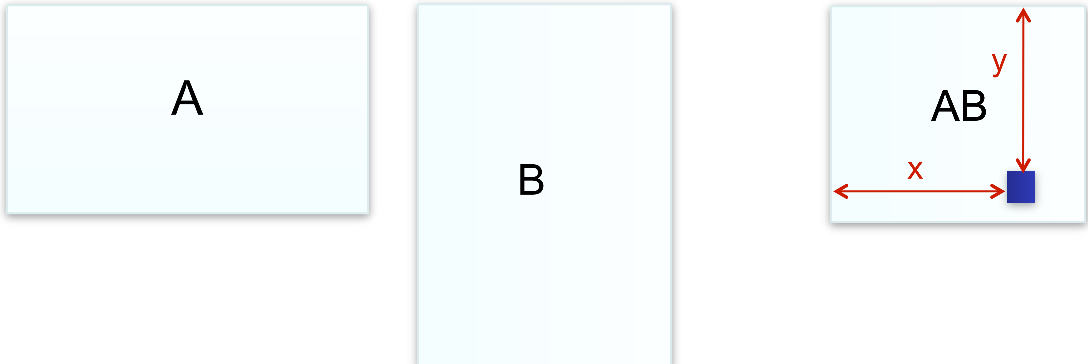
- Implicitly parallel description



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);
  }
}
```

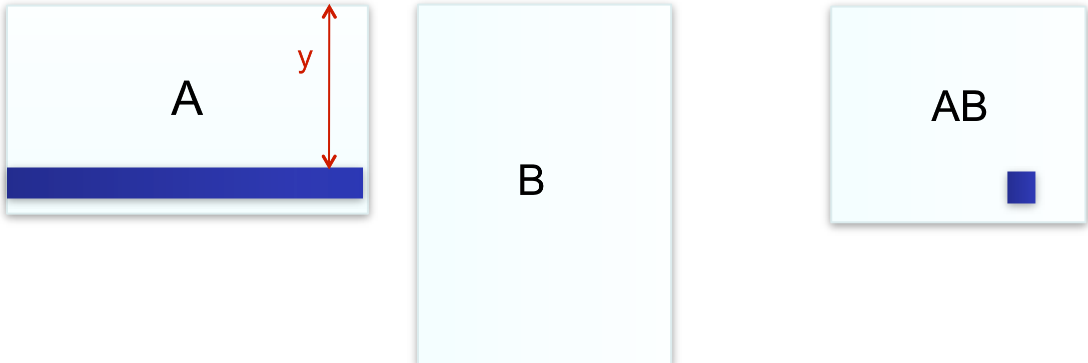
- Implicitly parallel description



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);
  }
}
```

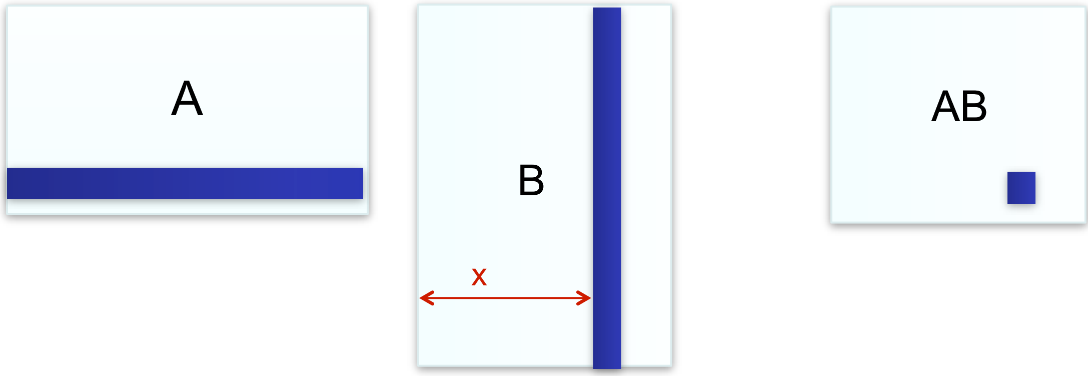
- Implicitly parallel description



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);
  }
}
```

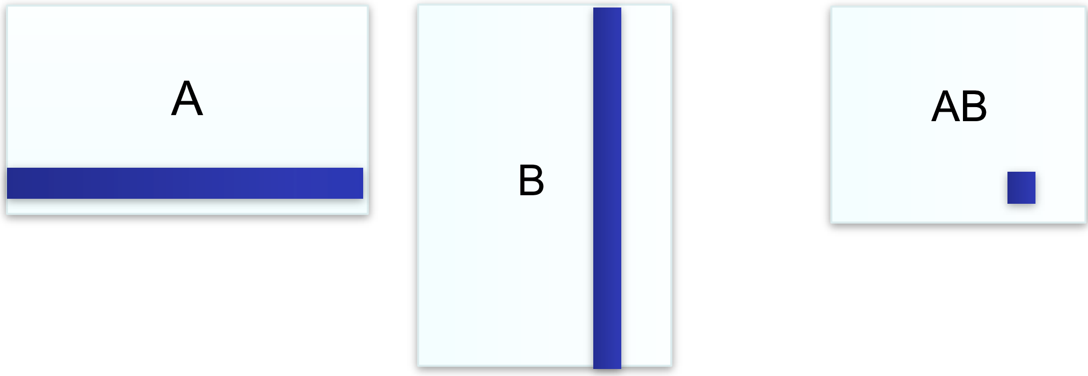
- Implicitly parallel description



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);
  }
}
```

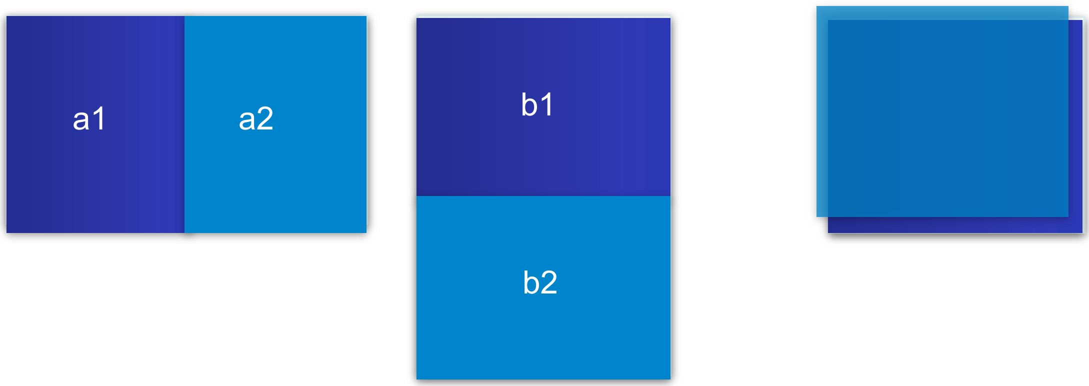
- Implicitly parallel description



PetaBricks Language

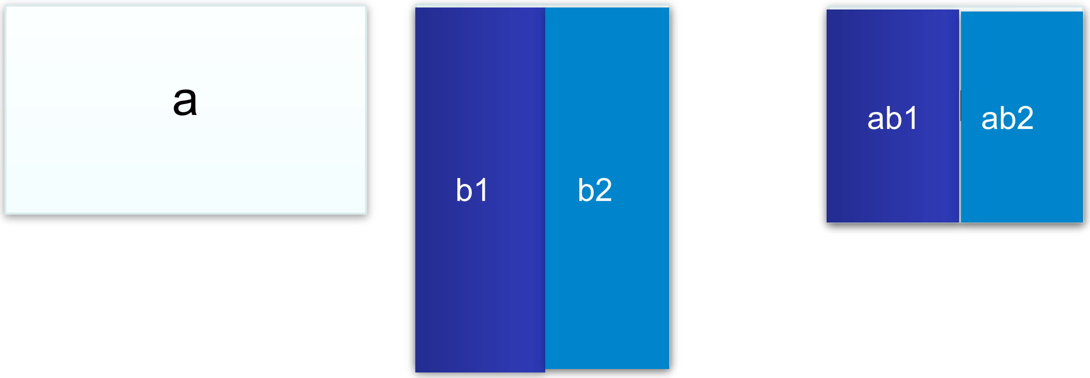
```
transform MatrixMultiply
from A[c,h], B[w,c]
to AB[w,h]
{
   // Base case, compute a single element
   to(AB.cell(x,y) out)
   from(A.row(y) a, B.column(x) b) {
       out = dot(a, b);
  }

  // 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 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);
  }
```



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

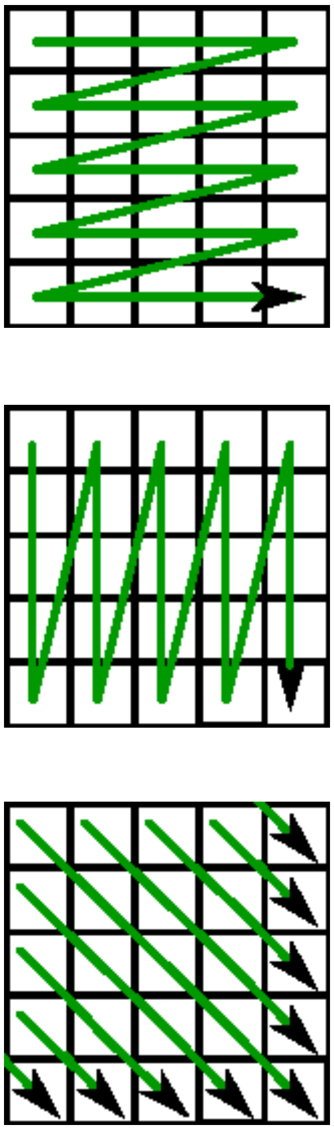
- Algorithmic choice is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
- Compiler re-use rules to create hybrid algorithms
- Can express choices at many different granularities

Synthesized Outer Control Flow

Synthesized Outer Control Flow

- Outer control flow synthesized by compiler

Synthesized Outer Control Flow



- Outer control flow synthesized by compiler
- Another choice that the programmer should not make

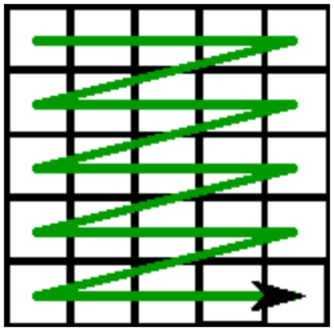
  By rows?

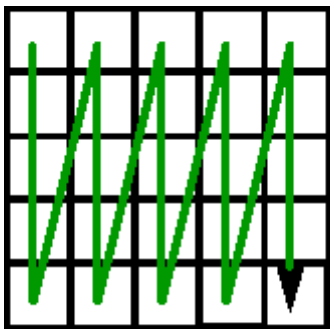
  By columns?

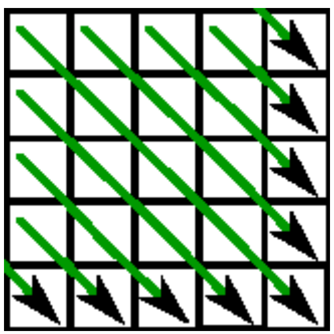
  Diagonal? Reverse order? Blocked?

  Parallel?

Synthesized Outer Control Flow







- Outer control flow synthesized by compiler
- Another choice that the programmer should not make

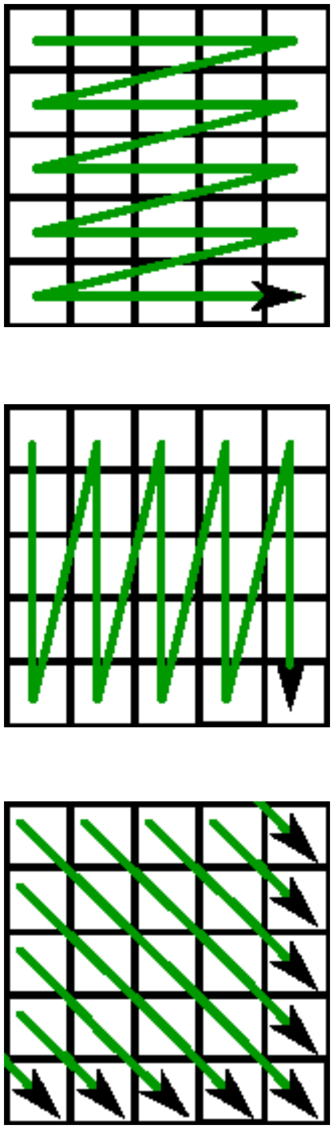
  By rows?

  By columns?

  Diagonal? Reverse order? Blocked?

  Parallel?
- Instead programmer provides explicit producer-consumer relations

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

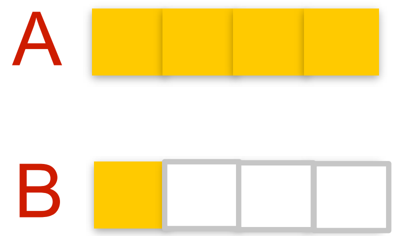
- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Another Example

```
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }

    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
```

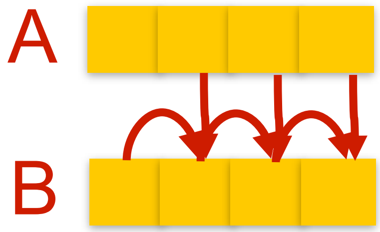
Another Example

```
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }

    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
```



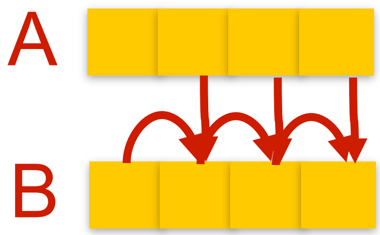
Another Example

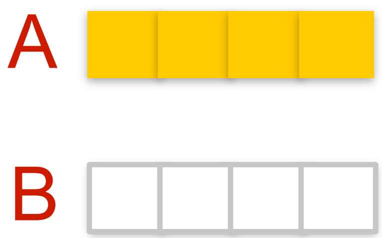
```
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }

    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
```



Another Example

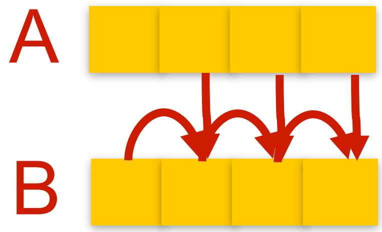
```
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }

    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
```

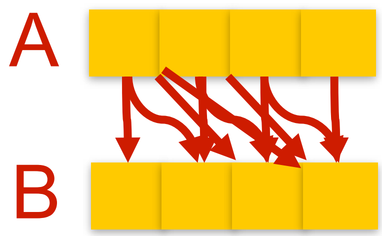




Another Example

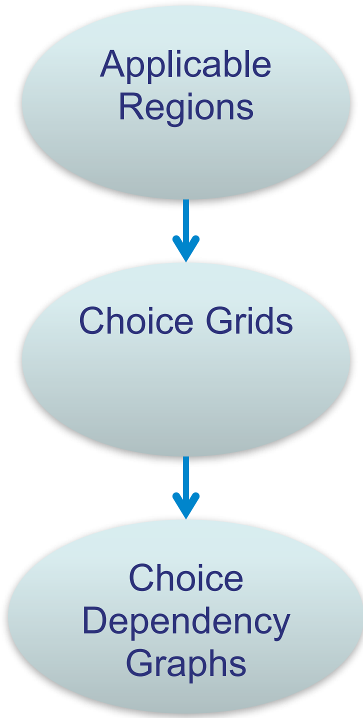
```
transform RollingSum
from A[n]
to B[n]
{
    // rule 0: use the previously computed value
    B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
        return a + leftSum;
    }

    // rule 1: sum all elements to the left
    B.cell(i) from (A.region(0, i) in) {
        return sum(in);
    }
}
```





Compilation Process

- Applicable Regions
- Choice Grids
- Choice Dependency Graphs

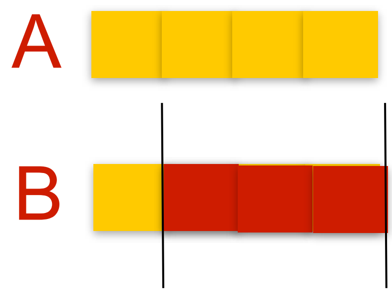


Applicable Regions

// rule 0: use the previously computed value

```
B.cell(i) from (A.cell(i) a, B.cell(i-1) leftSum) {
    return a + leftSum;
}
```

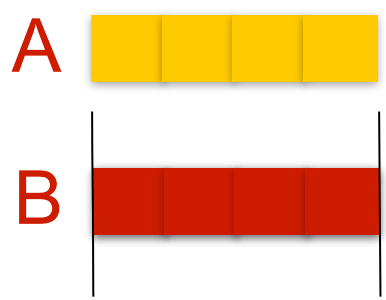
Applicable Region: $1 \le i < n$

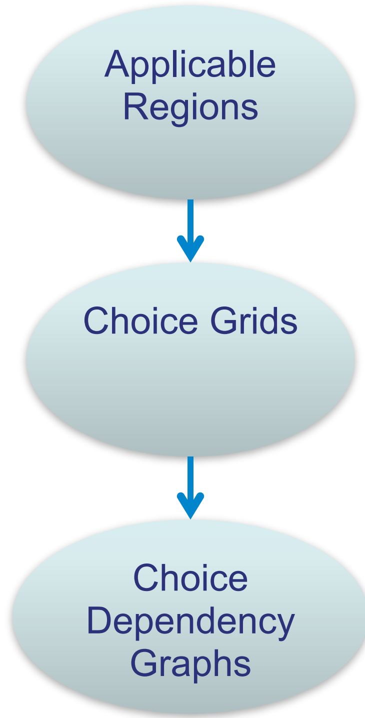


// rule 1: sum all elements to the left

```
B.cell(i) from (A.region(0, i) in) {
    return sum(in);
}
```

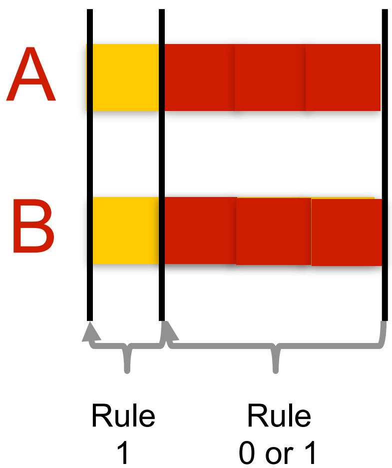
Applicable Region: $0 \le i < n$

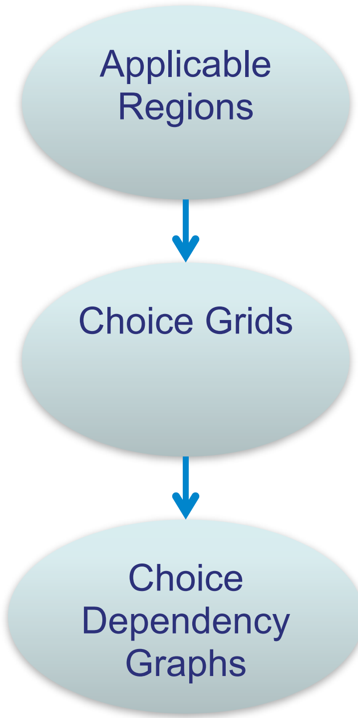




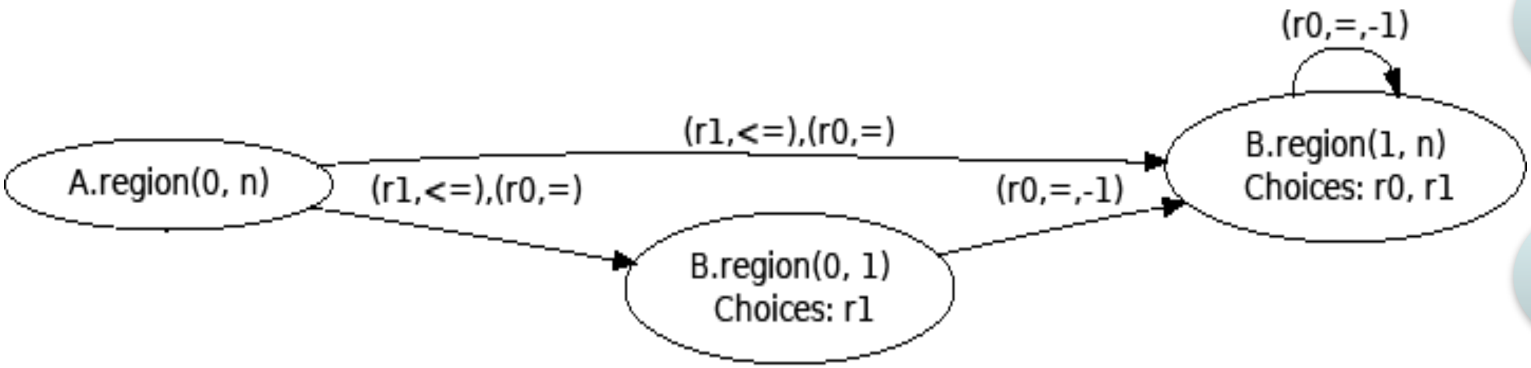
Choice Grids

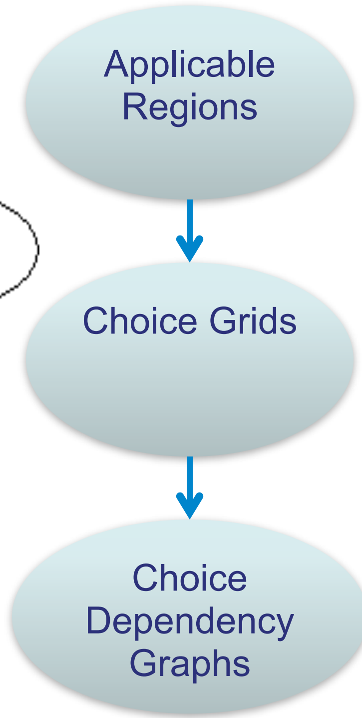
- Divide data space into symbolic regions with common sets of choices
- In this simple example:
  - A: Input (no choices)
  - B: [0; 1) = rule 1
  - B: [1; n) = rule 0 or rule 1
- Applicable regions map *rules* → *symbolic data*
- Choice grids map *symbolic data* → *rules*





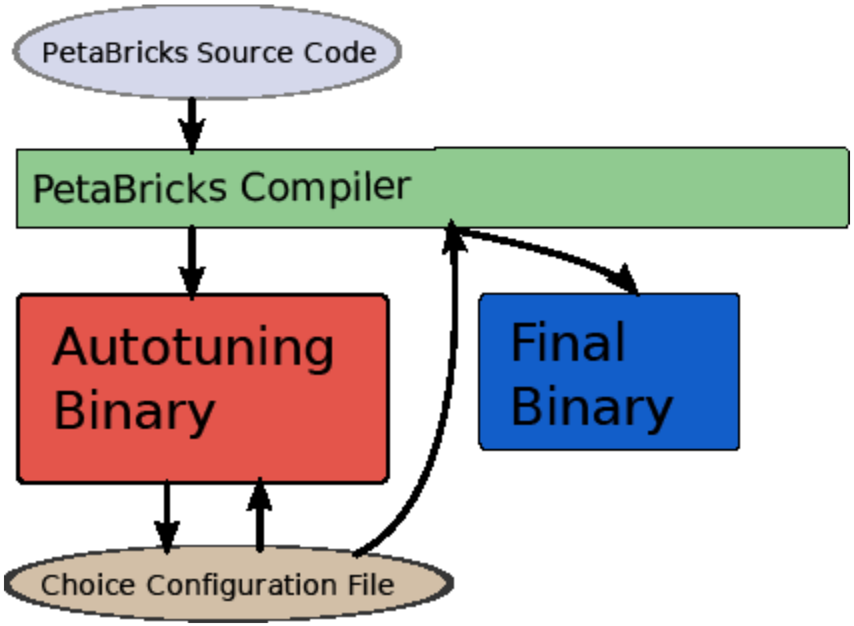
Choice Dependency Graphs





- Adds dependency edges between symbolic regions
- Edges annotated with directions and rules
- Many compiler passes on this IR to:
  - Simplify complex dependency patterns
  - Add choices

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

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Sort

Time

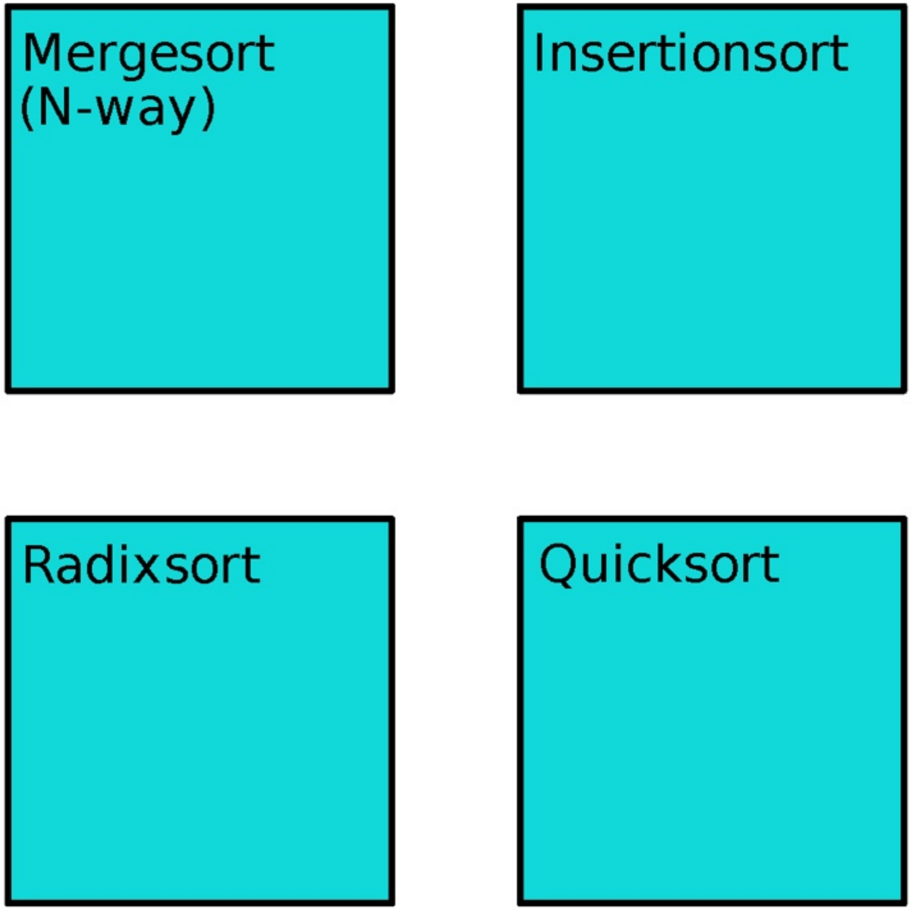
Size

Sort

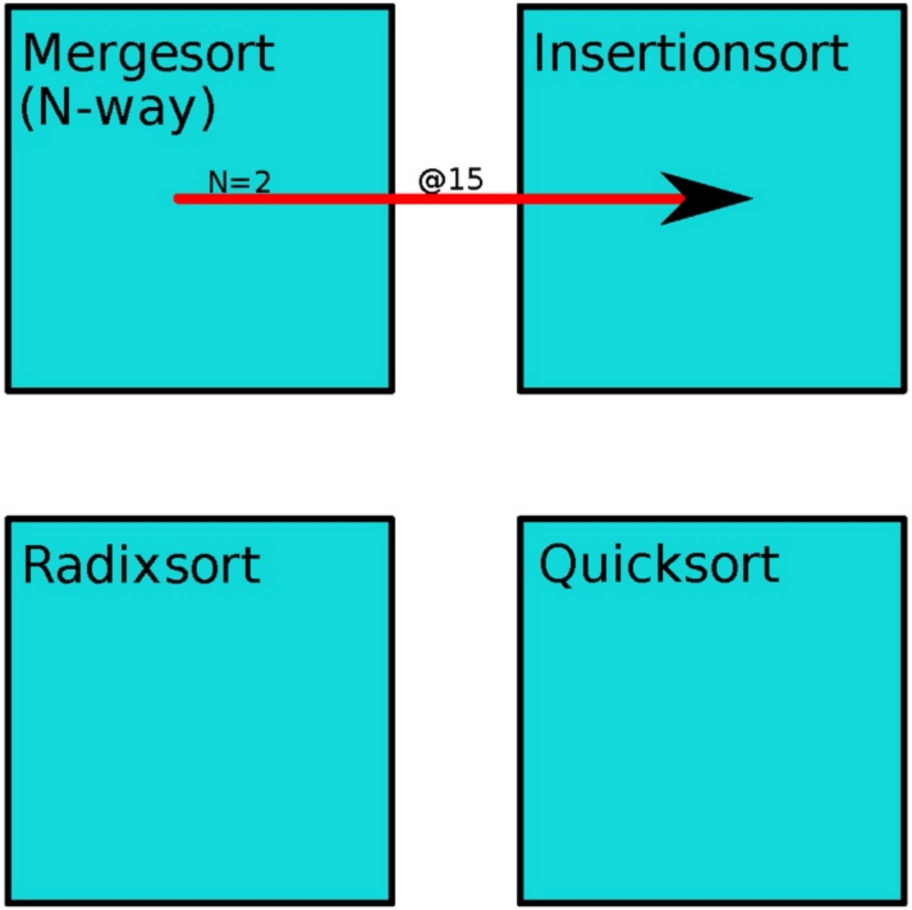
Time

Size

Algorithmic Choice in Sorting

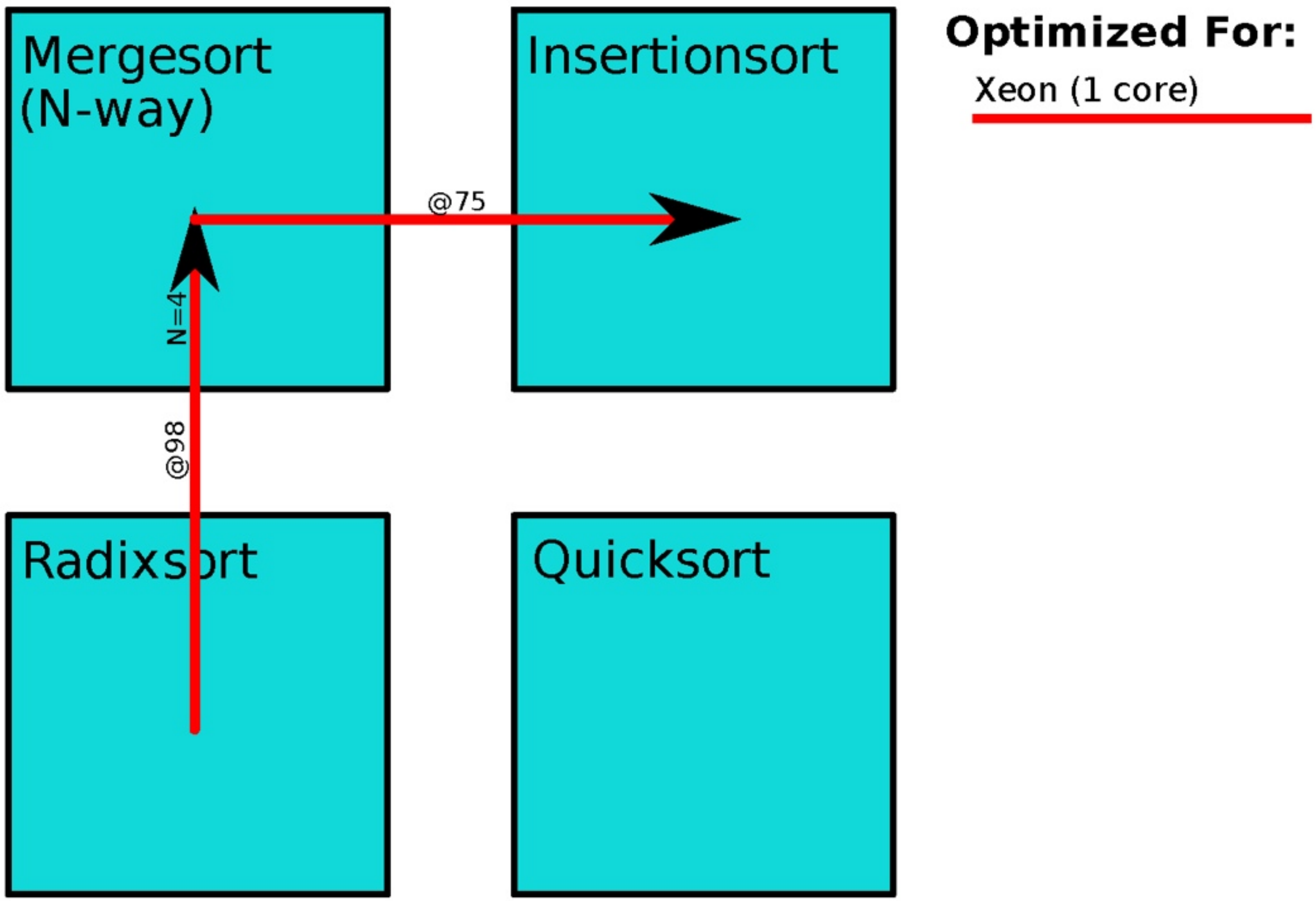


Algorithmic Choice in Sorting

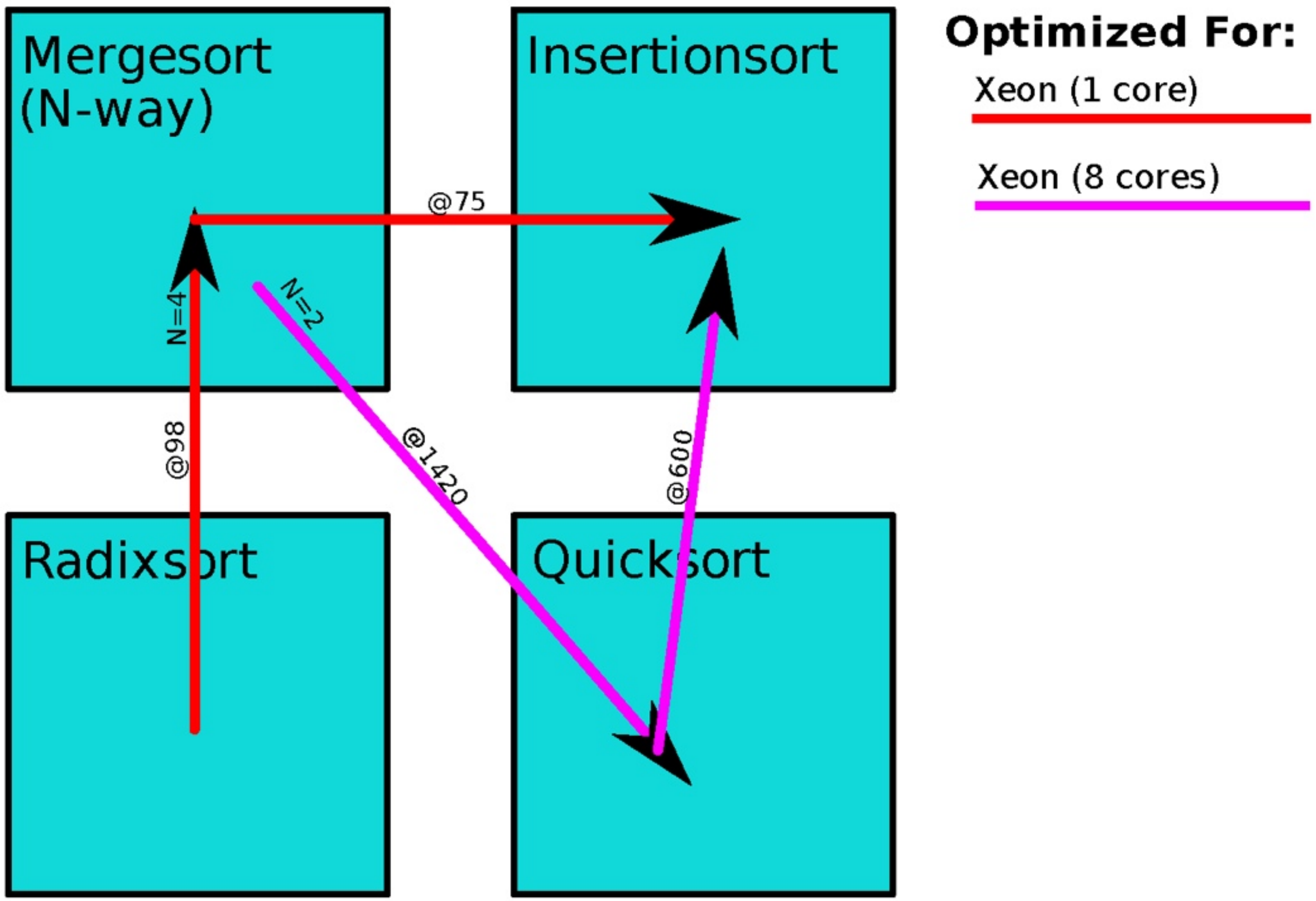


STL Algorithm

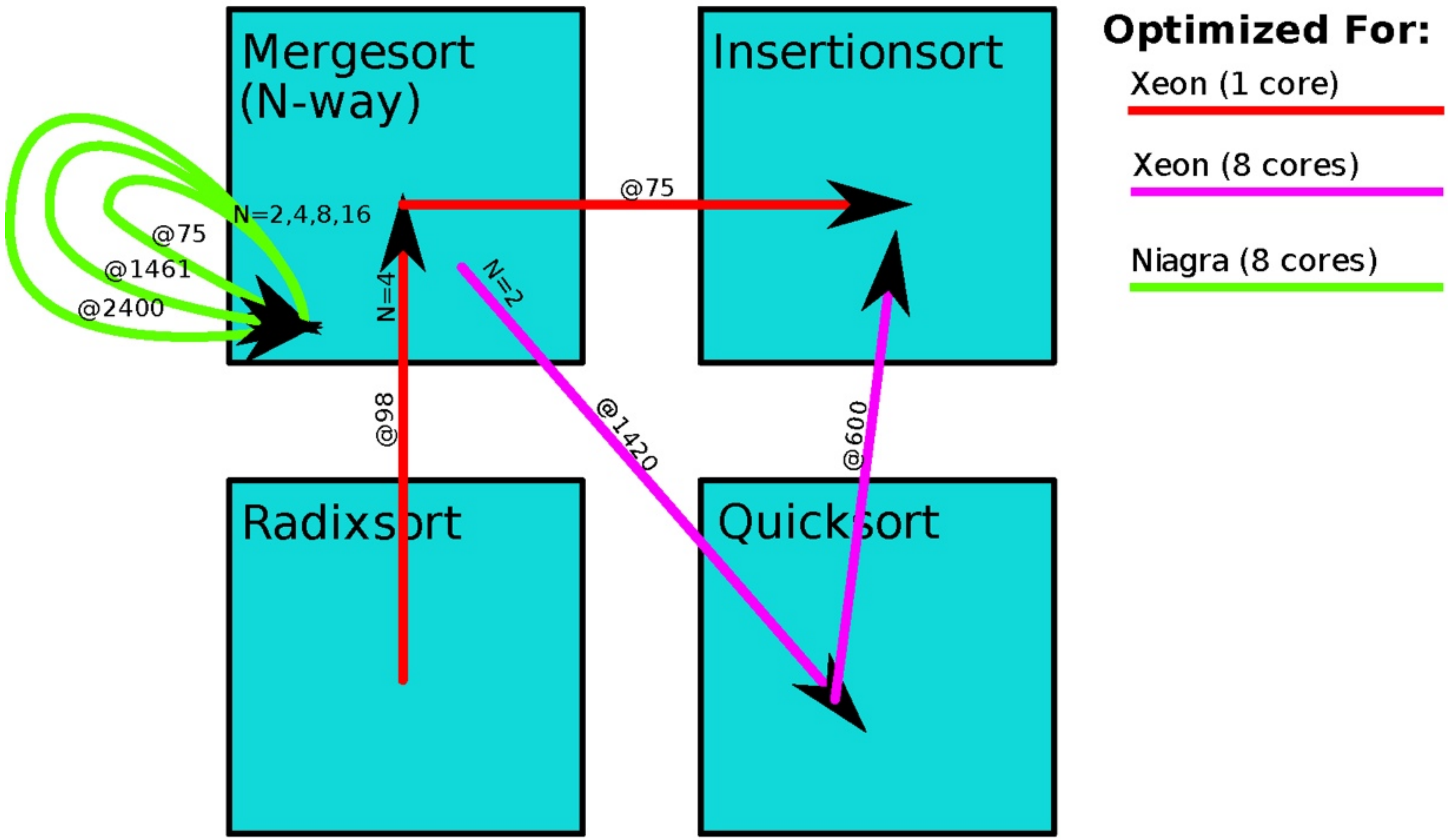
Algorithmic Choice in Sorting



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	-

Matrix Multiply

Time

Size

Matrix Multiply

Time

Size

Eigenvector Solve

Time

Size

Eigenvector Solve

Time

Size

Outline

- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Variable Accuracy Algorithms

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

A Very Brief Multigrid Intro

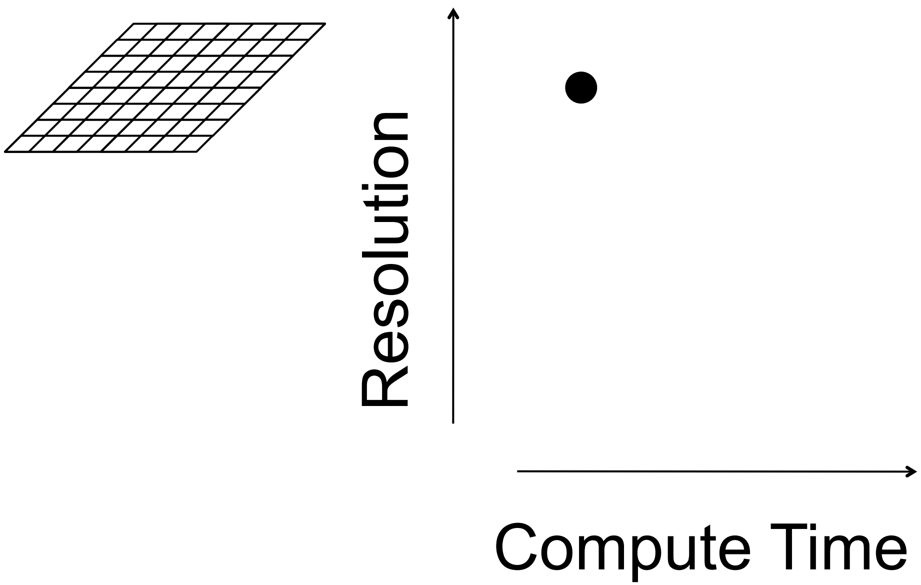
- Used to iteratively solve PDEs over a gridded domain
- **Relaxations** update points using neighboring values (stencil computations)

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

A Very Brief Multigrid Intro

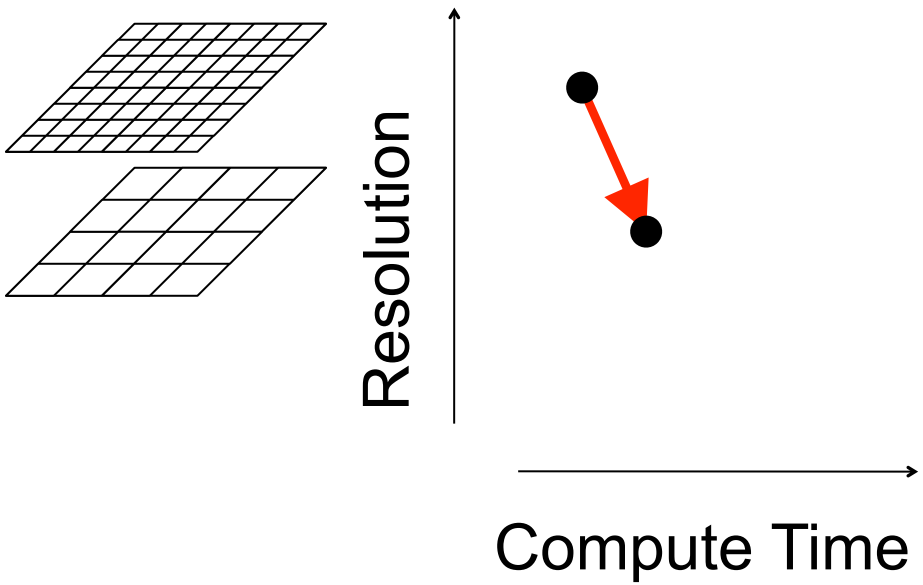
- Used to iteratively solve PDEs over a gridded domain
- **Relaxations** update points using neighboring values (stencil computations)
- **Restrictions** and **Interpolations** compute new grid with coarser or finer discretization



- Relax on current grid

A Very Brief Multigrid Intro

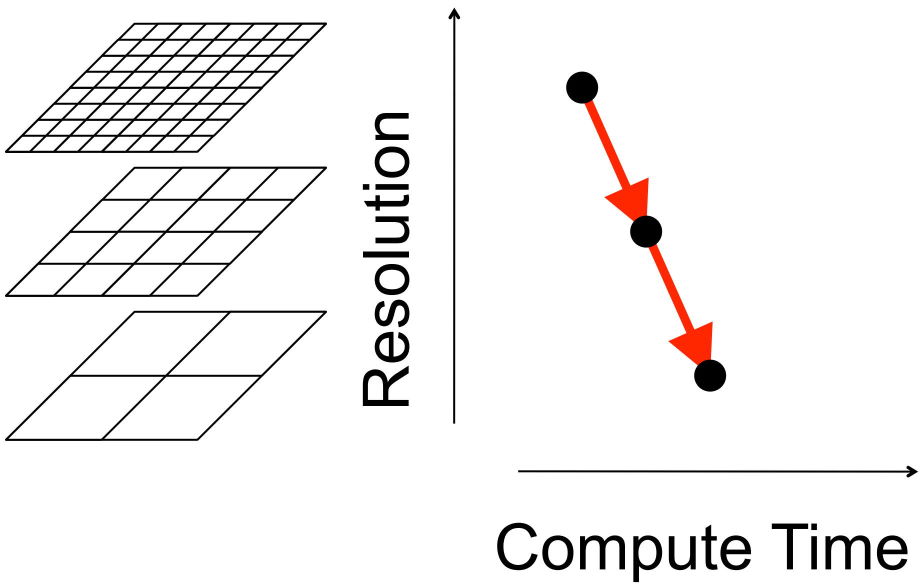
- Used to iteratively solve PDEs over a gridded domain
- **Relaxations** update points using neighboring values (stencil computations)
- **Restrictions** and **Interpolations** compute new grid with coarser or finer discretization



- Relax on current grid
- ↘ Restrict to coarser grid

A Very Brief Multigrid Intro

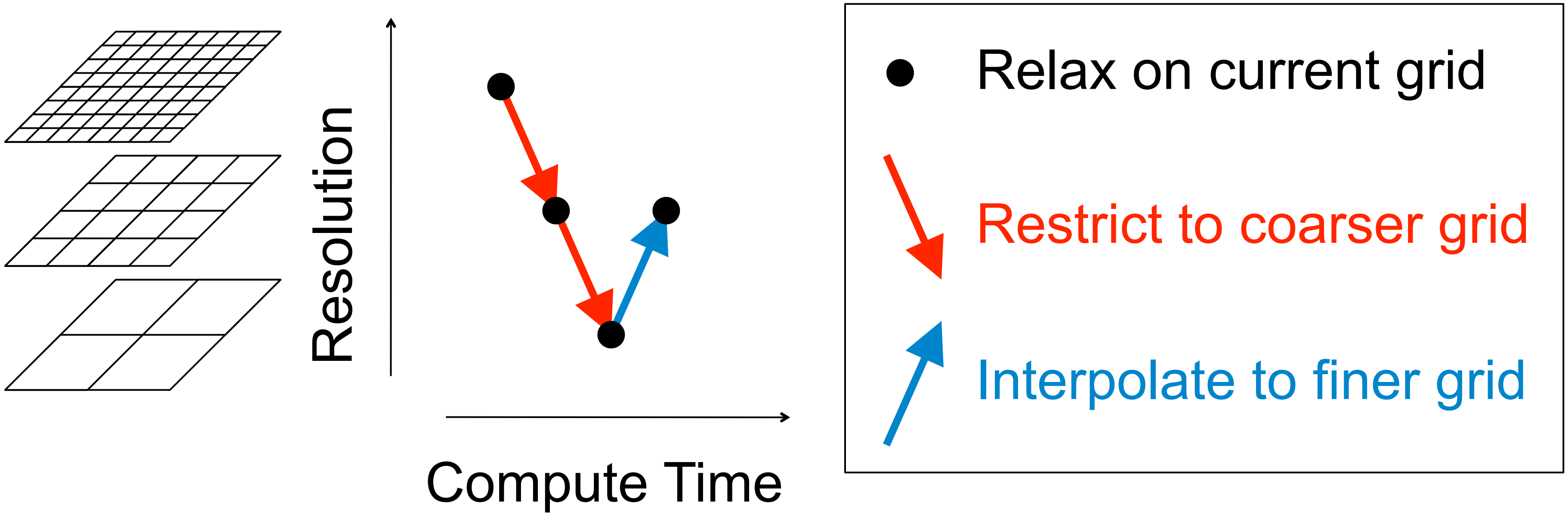
- Used to iteratively solve PDEs over a gridded domain
- **Relaxations** update points using neighboring values (stencil computations)
- **Restrictions** and **Interpolations** compute new grid with coarser or finer discretization



- Relax on current grid
- Restrict to coarser grid

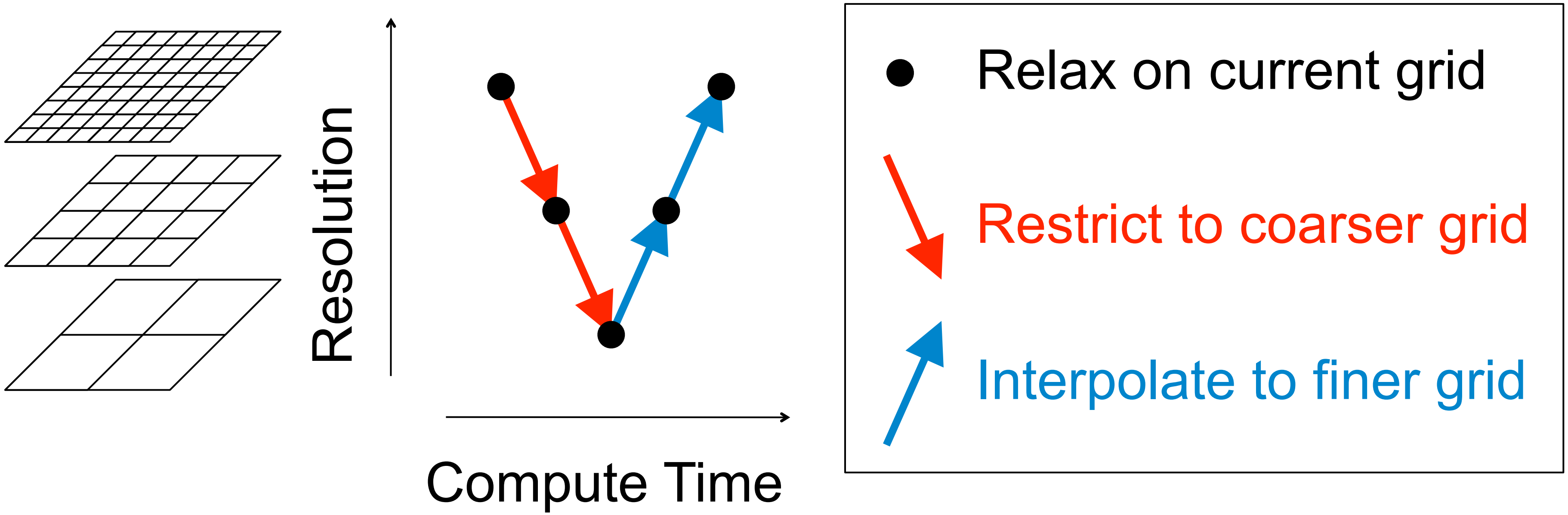
A Very Brief Multigrid Intro

- 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

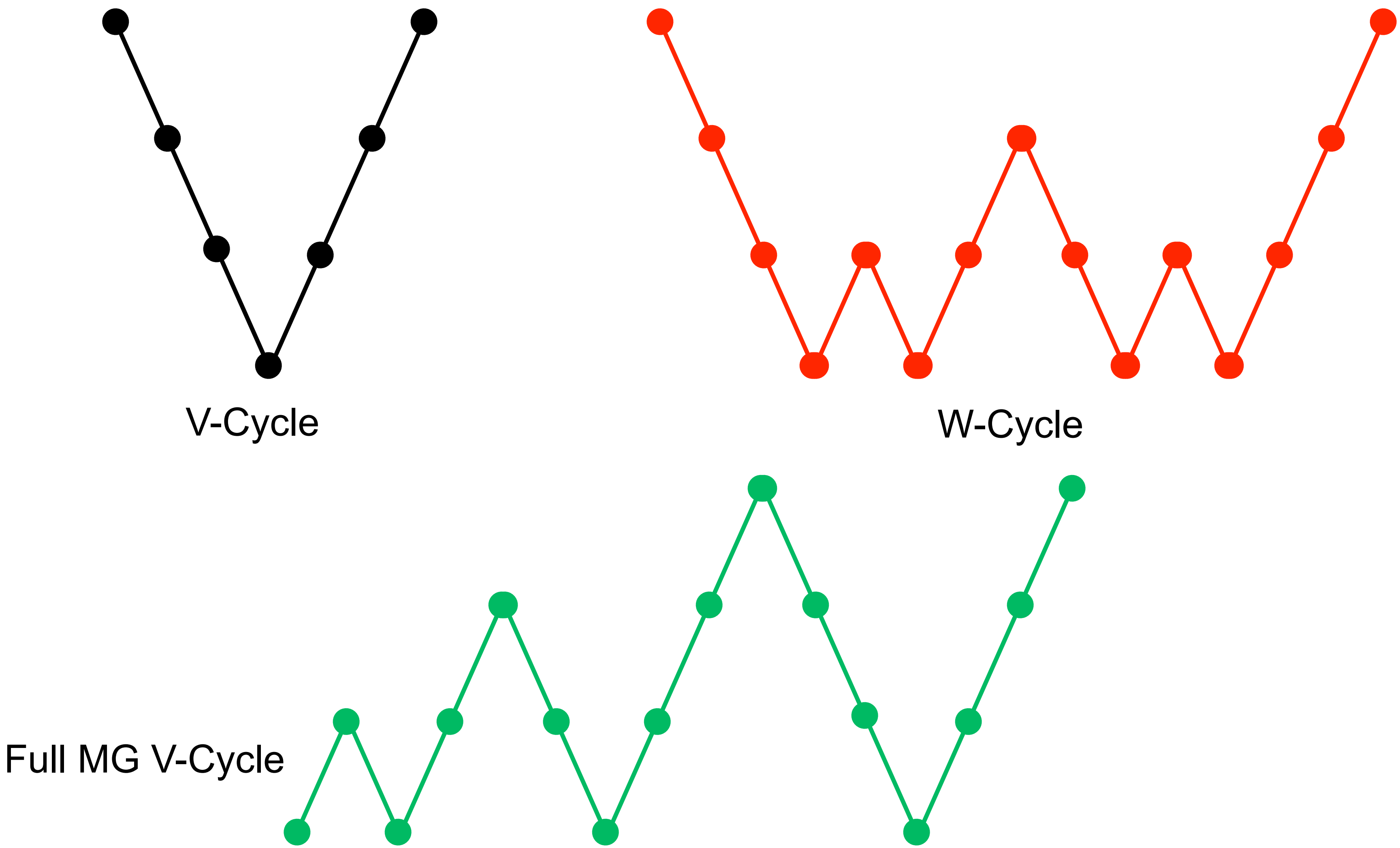


A Very Brief Multigrid Intro

- 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

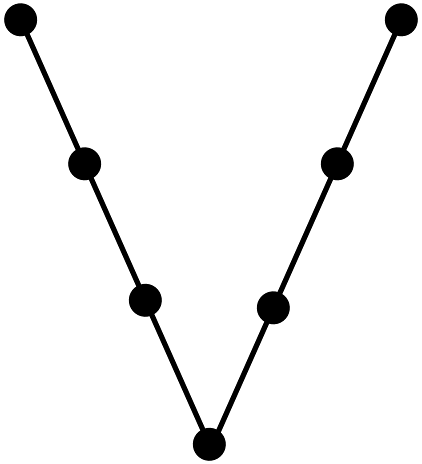


Multigrid Cycles

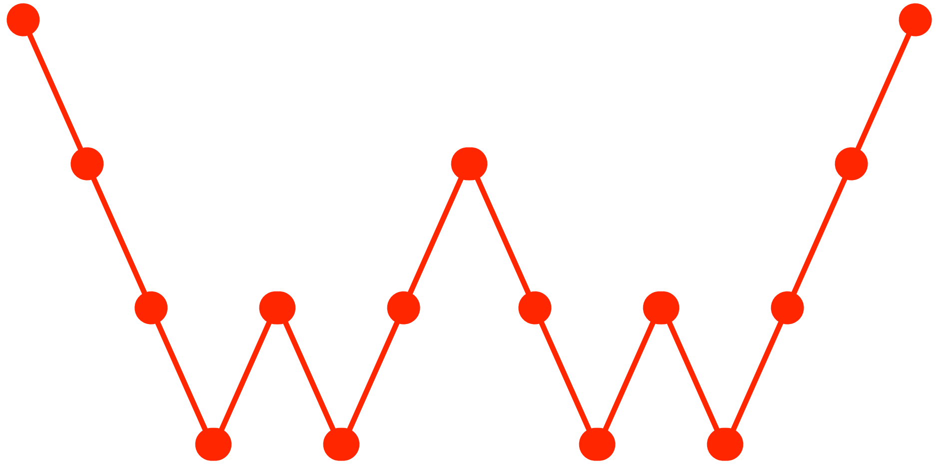


Standard Approaches

Multigrid Cycles



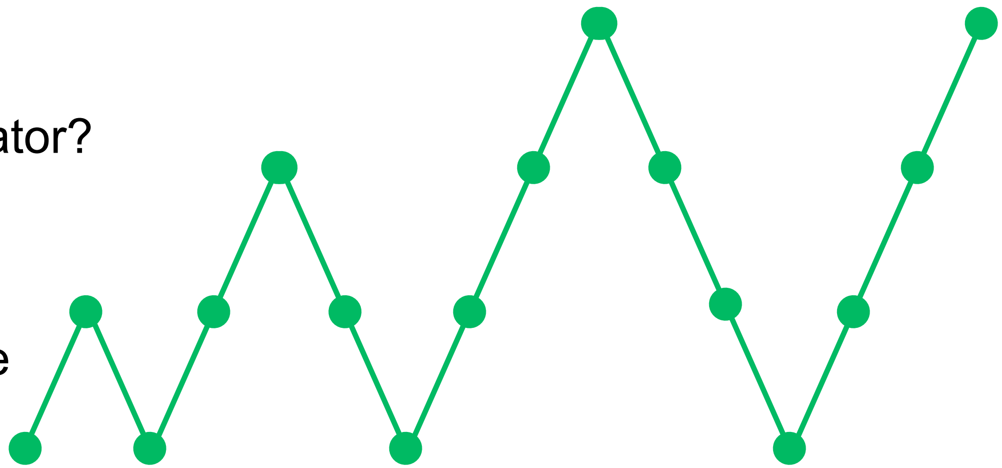
V-Cycle



W-Cycle

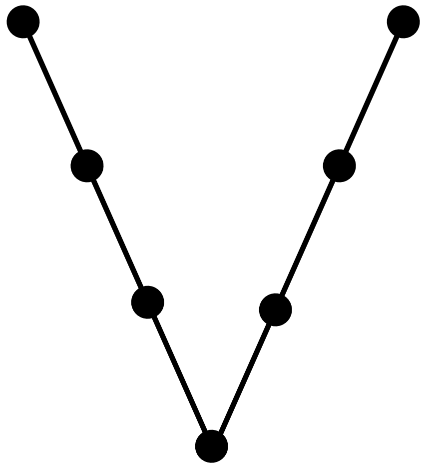
Relaxation operator?

Full MG V-Cycle

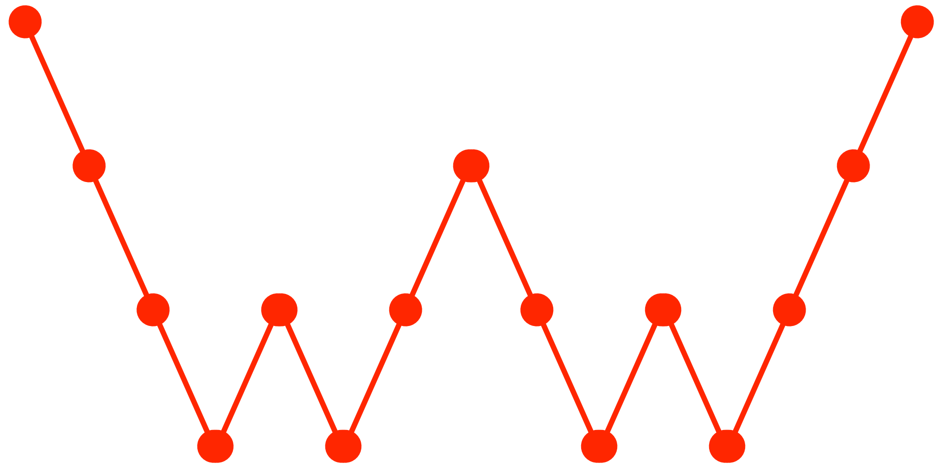


Standard Approaches

Multigrid Cycles

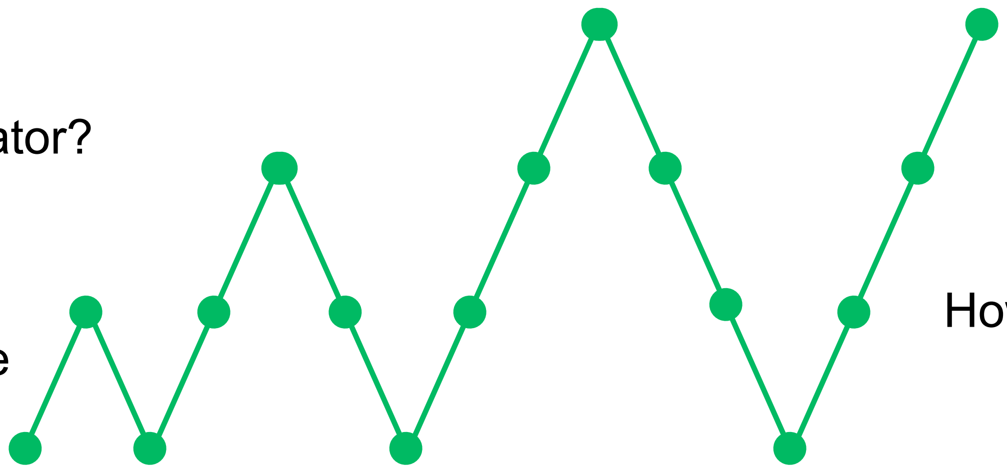


V-Cycle



W-Cycle

Relaxation operator?

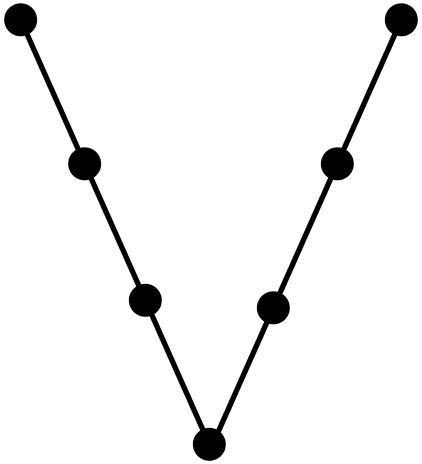


Full MG V-Cycle

How many iterations?

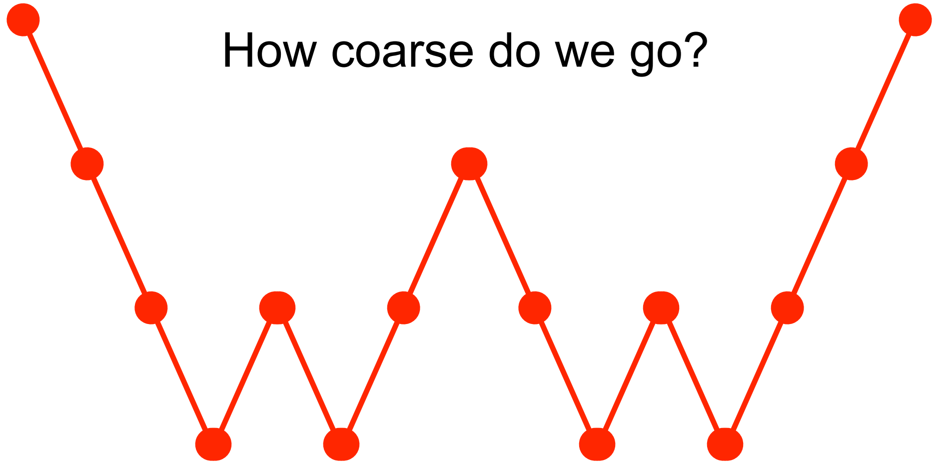
Standard Approaches

Multigrid Cycles



V-Cycle

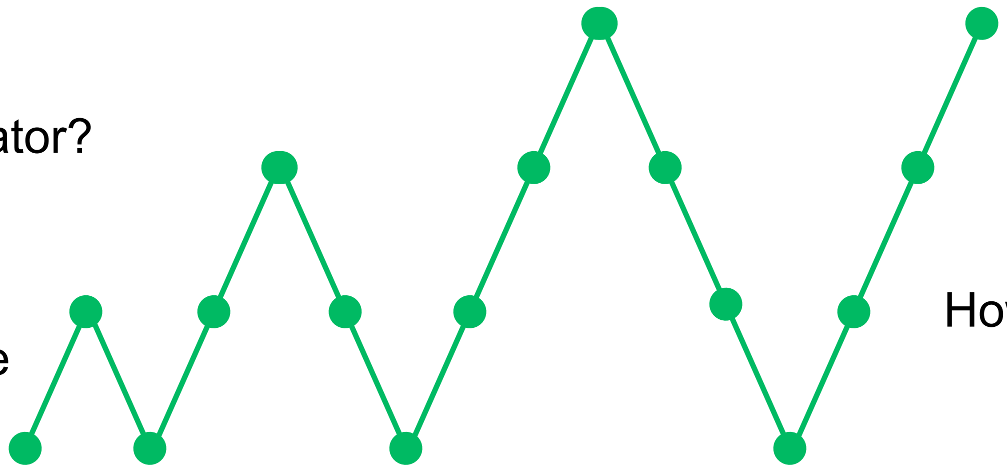
How coarse do we go?



W-Cycle

Relaxation operator?

Full MG V-Cycle

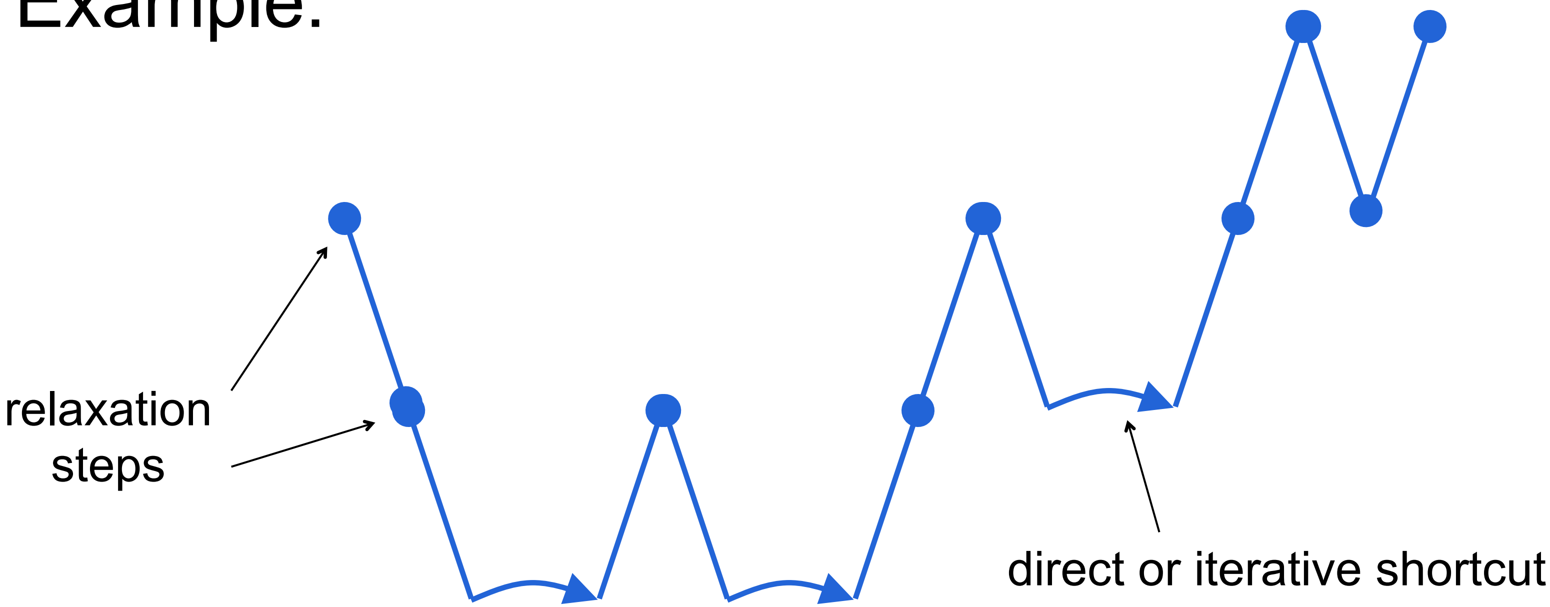


How many iterations?

Standard Approaches

Multigrid Cycles

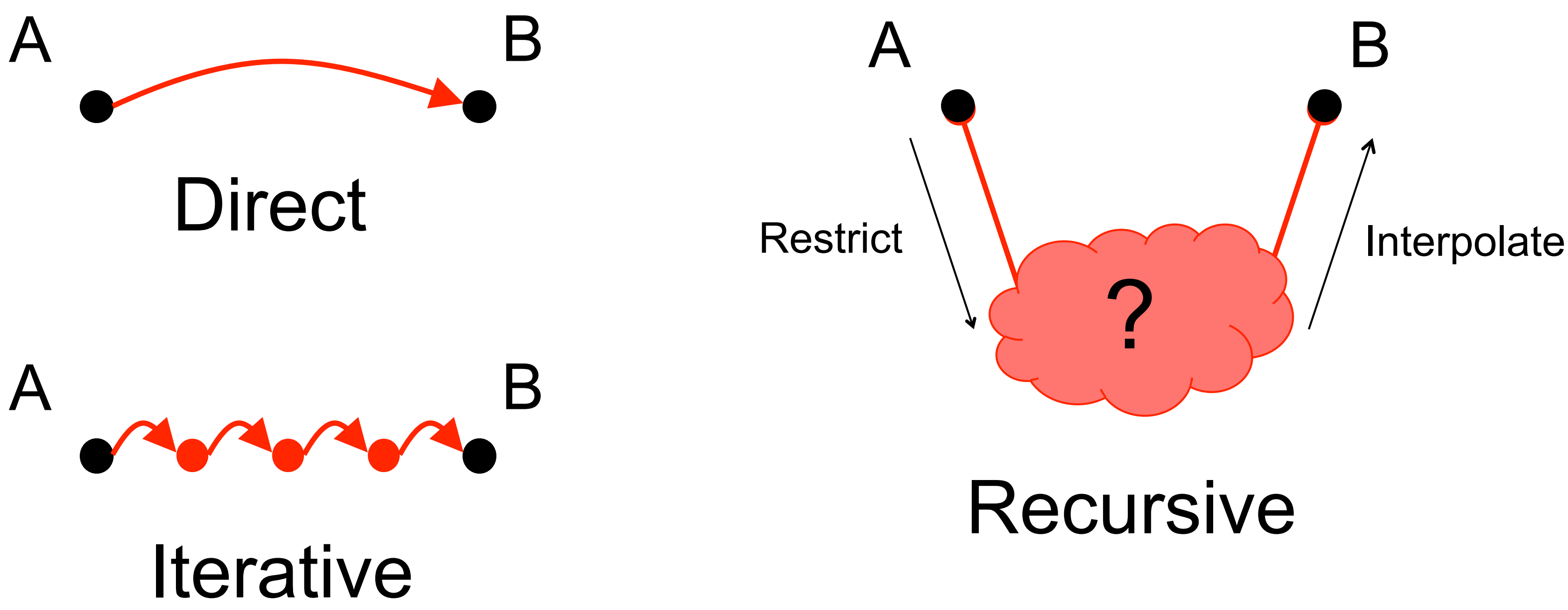
- Generalize the idea of what a multigrid cycle can look like
- Example:



- Goal: Auto-tune cycle shape for specific usage

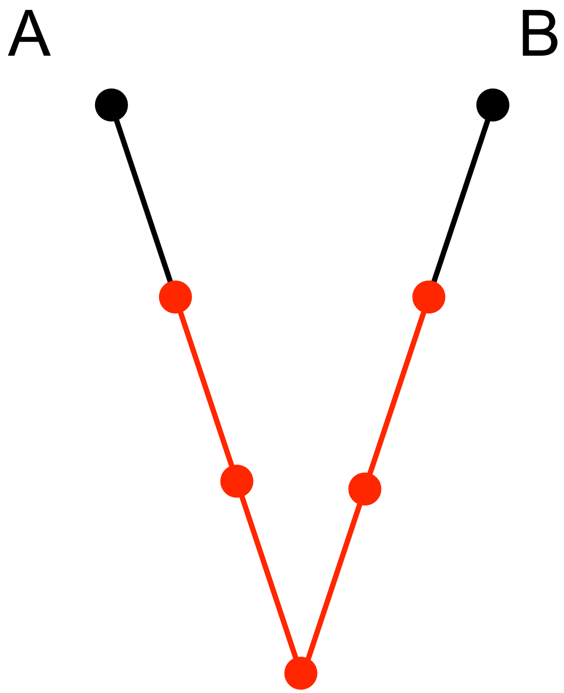
Algorithmic Choice in Multigrid

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



Algorithmic Choice in Multigrid

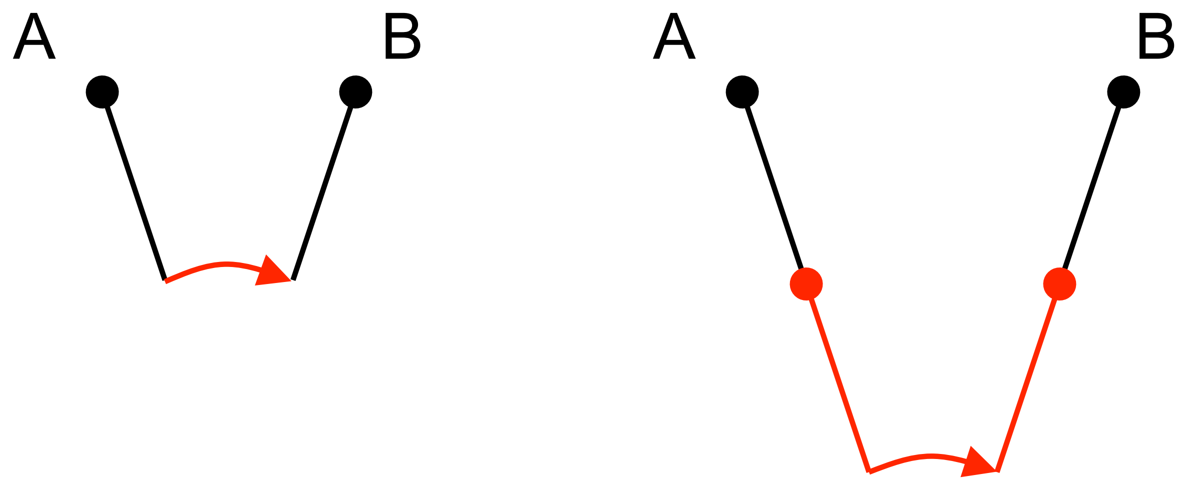
- Tuning cycle shape!
  - Examples of recursive options:



Standard V-cycle

Algorithmic Choice in Multigrid

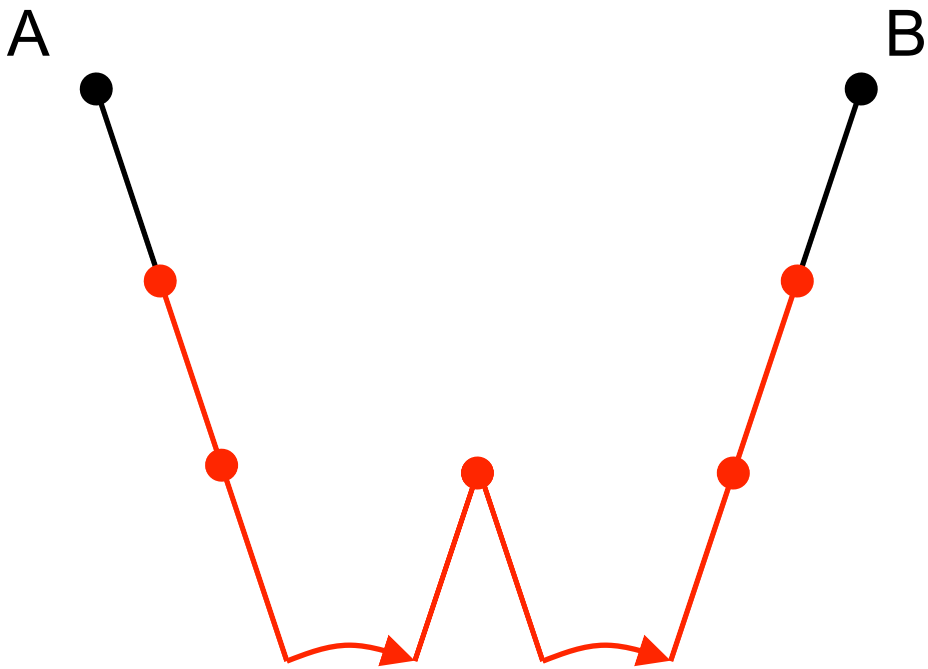
- Tuning cycle shape!
  - Examples of recursive options:



Take a shortcut at a coarser resolution

Algorithmic Choice in Multigrid

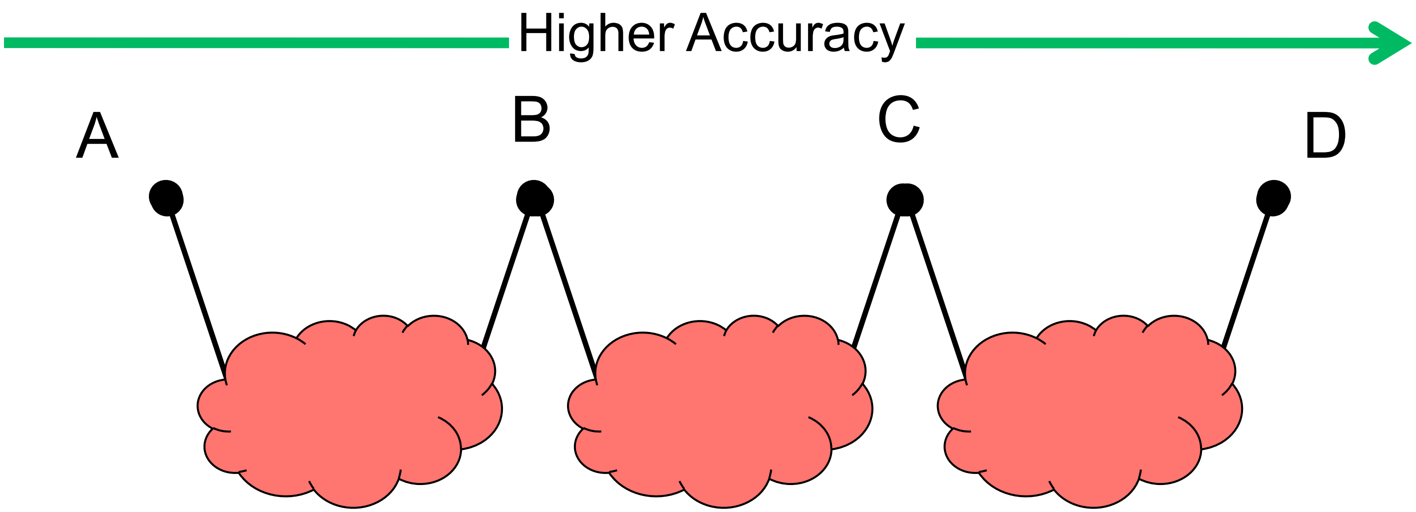
- Tuning cycle shape!
  - Examples of recursive options:



Iterating with shortcuts

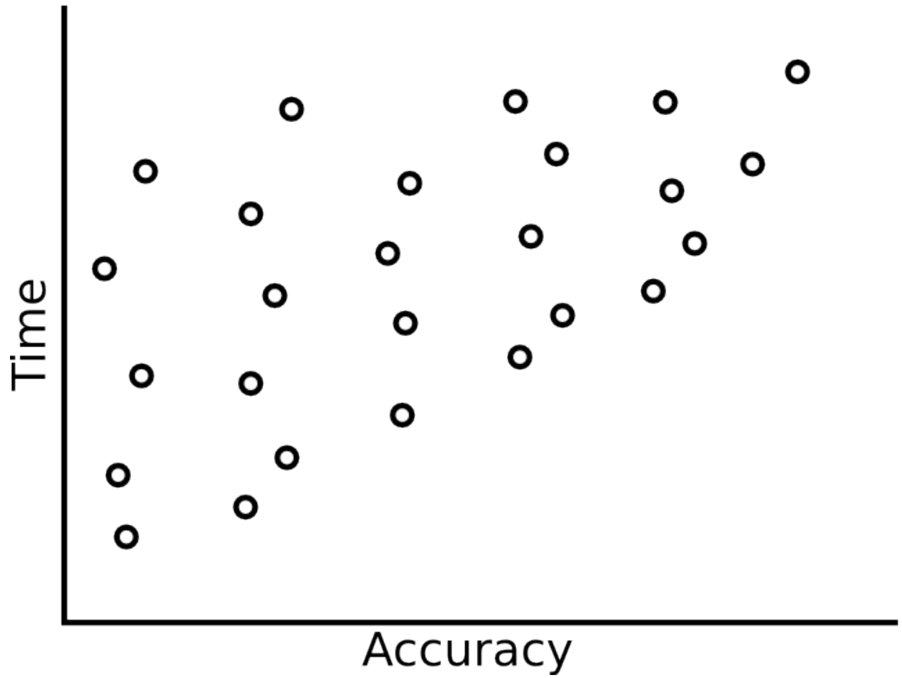
Algorithmic Choice in Multigrid

- Tuning cycle shape!
  - Once we pick a recursive option, how many times do we iterate?

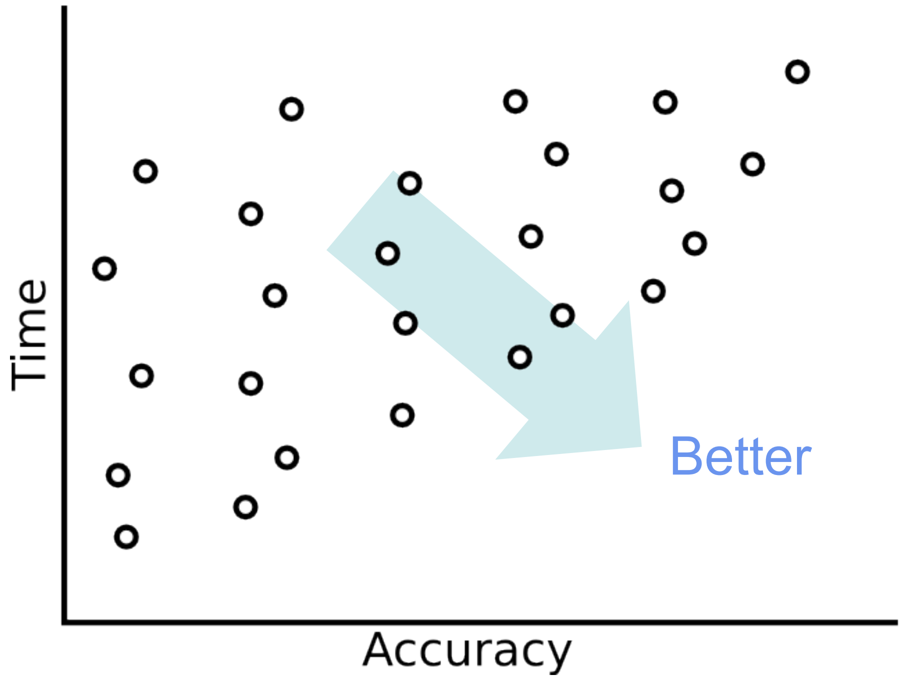


- Number of iterations depends on what **accuracy** we want at the current grid resolution!

Optimal Subproblems

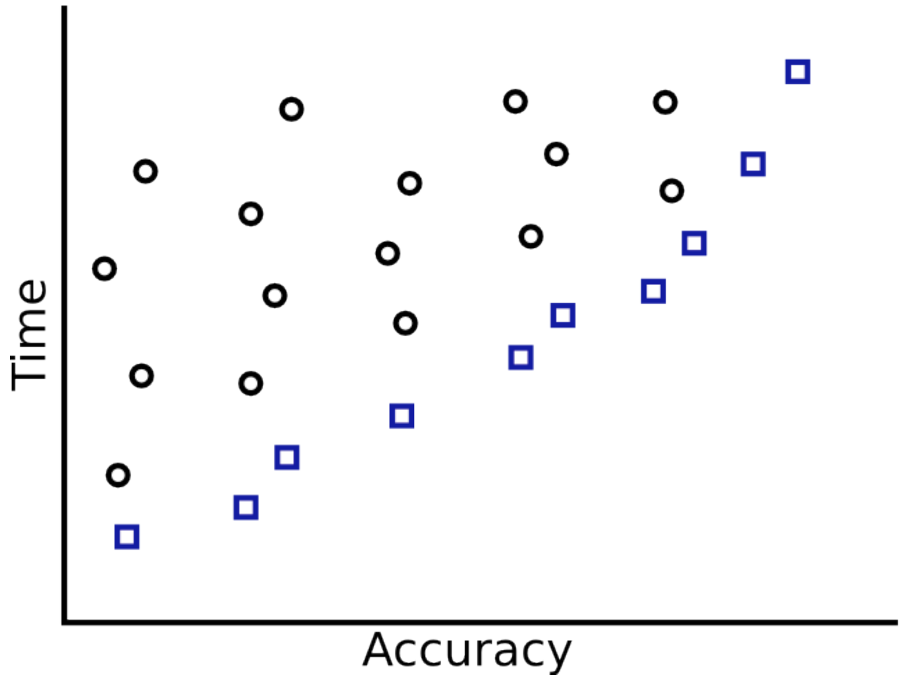


Optimal Subproblems



Optimal Subproblems

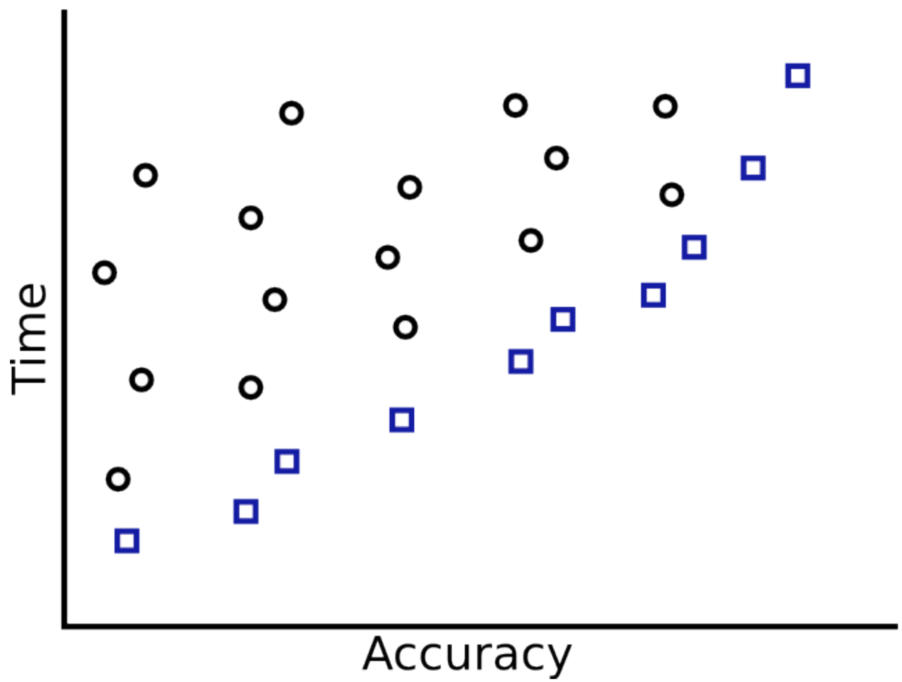
- Plot all cycle shapes for a given grid resolution:



Keep only the optimal ones!

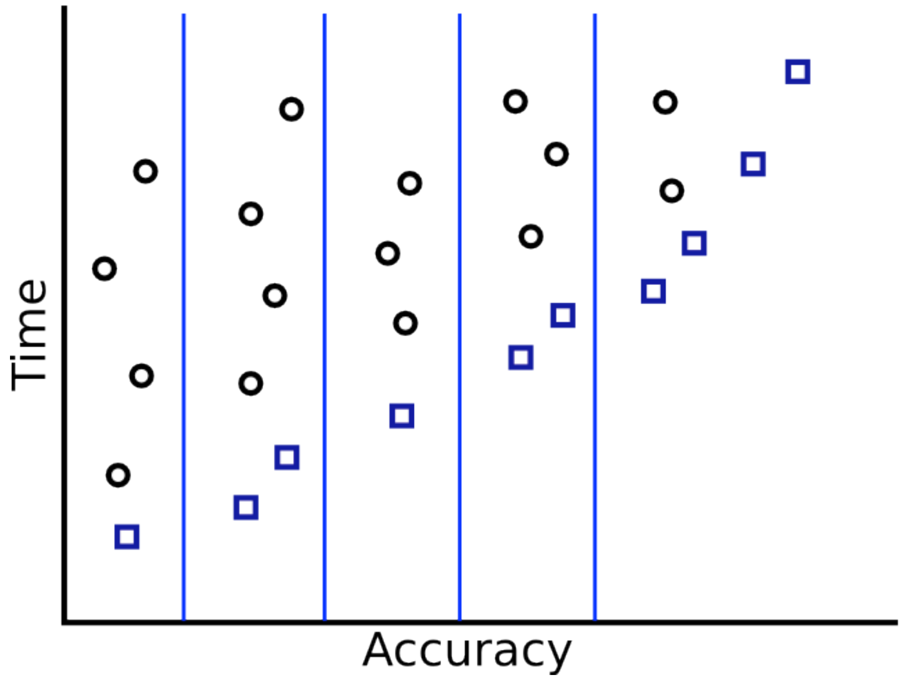
- Idea: Maintain a **family** of optimal algorithms for each grid resolution

The Discrete Solution



The Discrete Solution

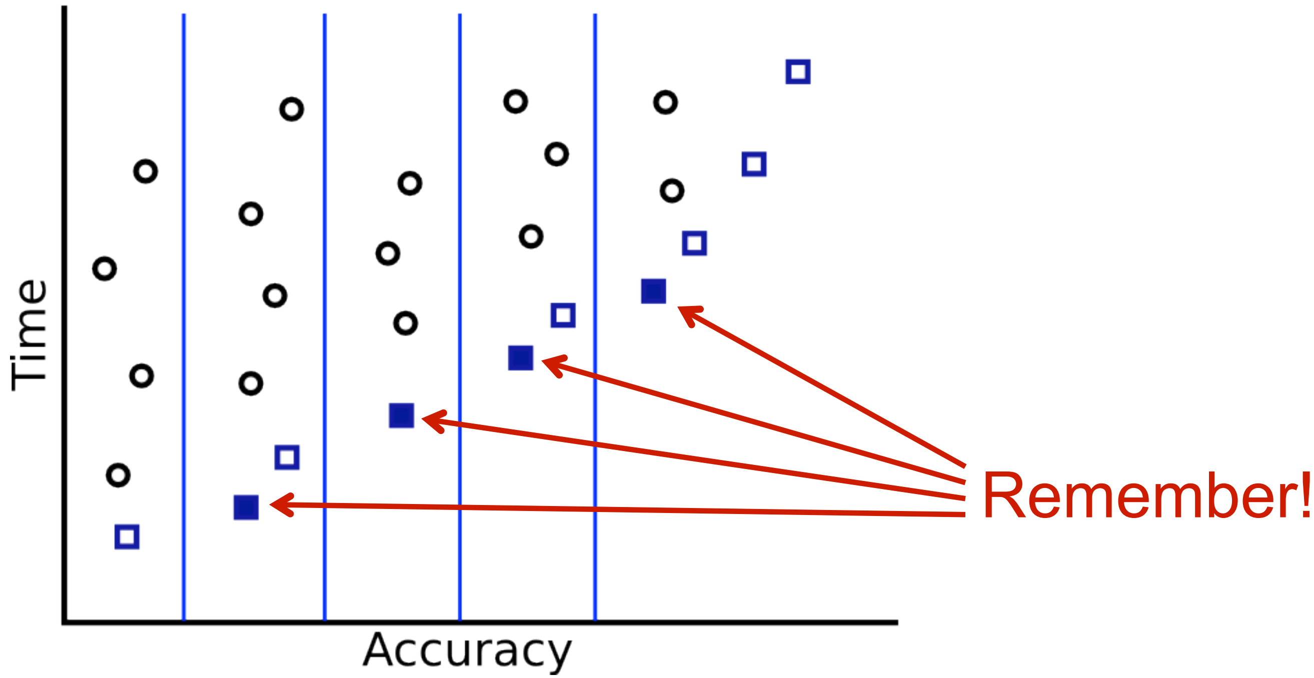
- Problem: Too many optimal cycle shapes to remember



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

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

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

Use Dynamic Programming

- 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

Use Dynamic Programming

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

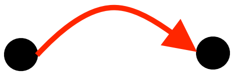
- Allow shortcuts to stop recursion early
- Allow multiple iterations of sub-cycles to explore time vs. accuracy space

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

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

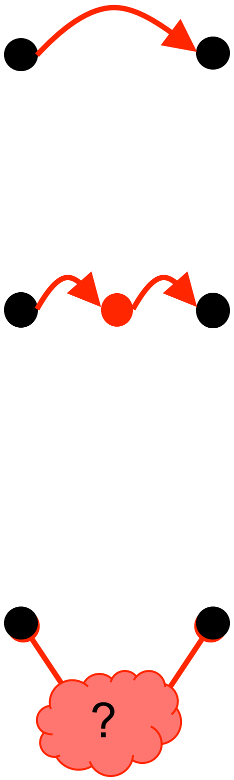
OR

// 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

Auto-tuning the V-cycle

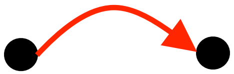


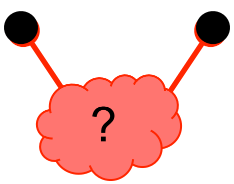

```
transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
{
    // Base case
    // Direct solve

        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

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
}
```

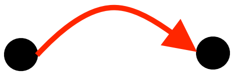
- Algorithmic choice

  Shortcut base cases

  Recursively call some optimized sub-cycle

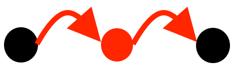
Auto-tuning the V-cycle

transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
{



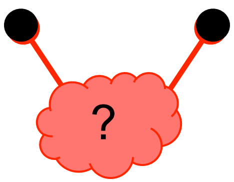
// Base case
// Direct solve

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

Variable Accuracy Keywords

transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]

Variable Accuracy Keywords

- **accuracy_variable** – tunable variable

transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
accuracy_variable numIterations

Variable Accuracy Keywords

- **accuracy_variable** – tunable variable
- **accuracy_metric** – returns accuracy of output

transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
accuracy_variable numIterations
accuracy_metric Poisson2D_metric

Variable Accuracy Keywords

- **accuracy_variable** – tunable variable
- **accuracy_metric** – returns accuracy of output
- **accuracy_bins** – set of discrete accuracy bins

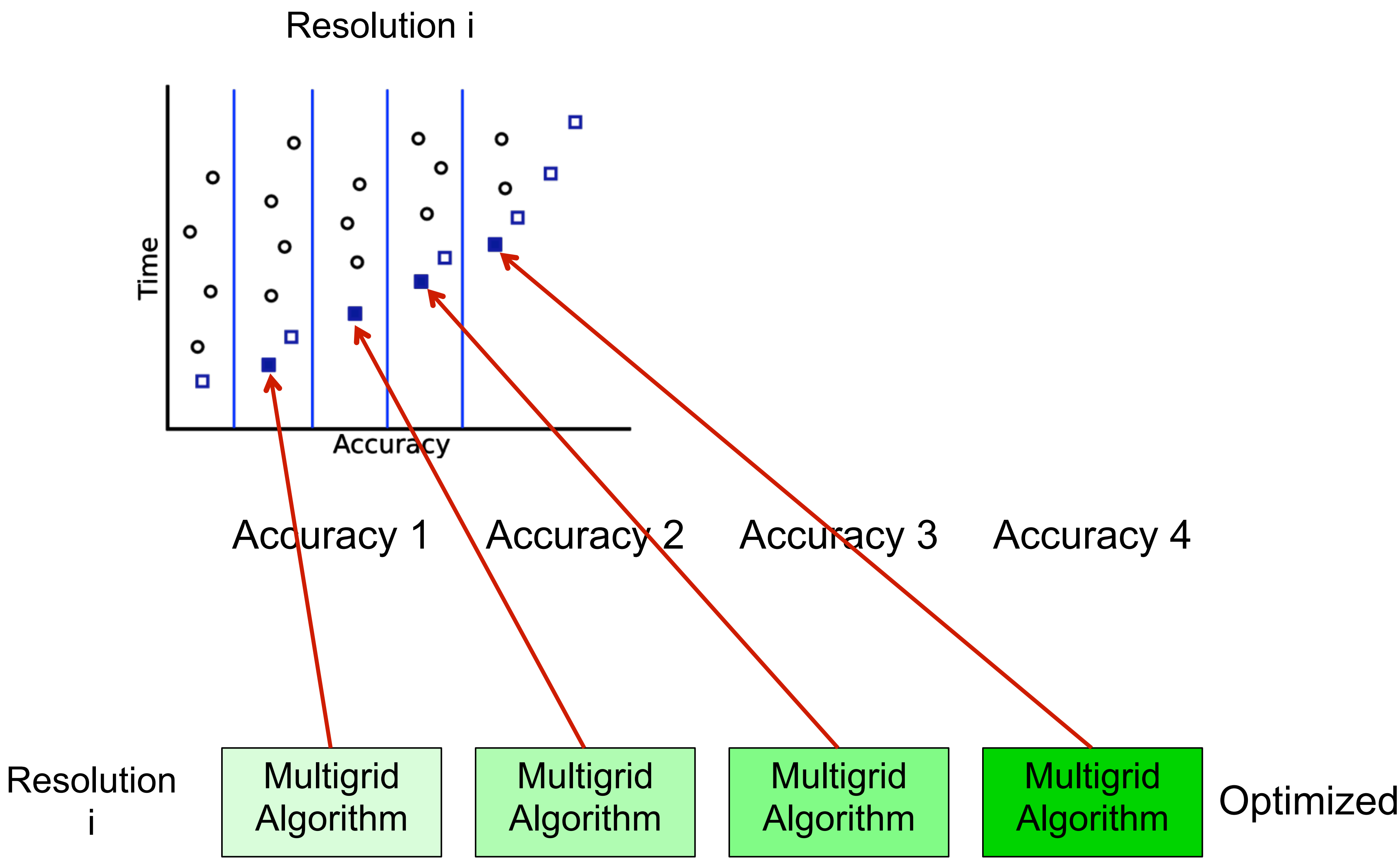
transform Multigrid_k
from X[n,n], B[n,n]
to Y[n,n]
accuracy_variable numIterations
accuracy_metric Poisson2D_metric
accuracy_bins 1e1 1e3 1e5 1e7

Variable Accuracy Keywords

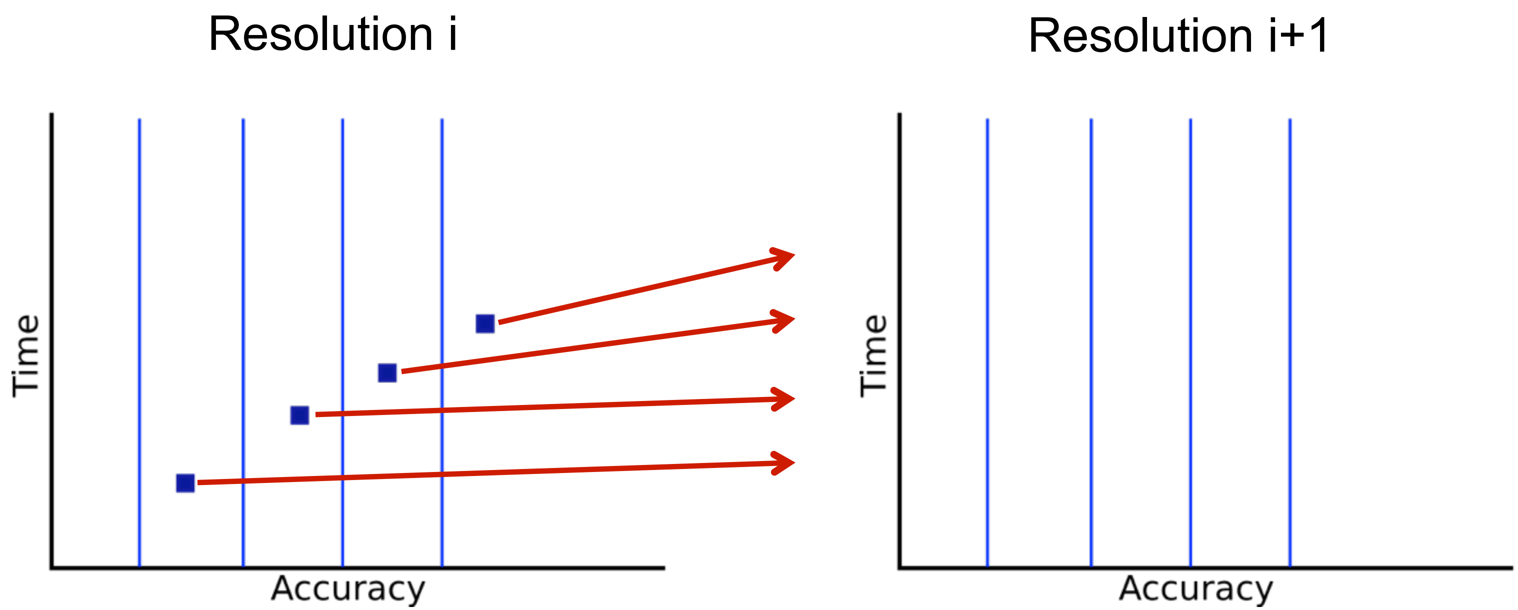
- **accuracy_variable** – tunable variable
- **accuracy_metric** – returns accuracy of output
- **accuracy_bins** – set of discrete accuracy bins
- **generator** – creates random inputs for accuracy measurement

transform Multigrid$_k$
from X[n,n], B[n,n]
to Y[n,n]
accuracy_variable numIterations
accuracy_metric Poisson2D_metric
accuracy_bins 1e1 1e3 1e5 1e7
generator Poisson2D_Generator

Training the Discrete Solution

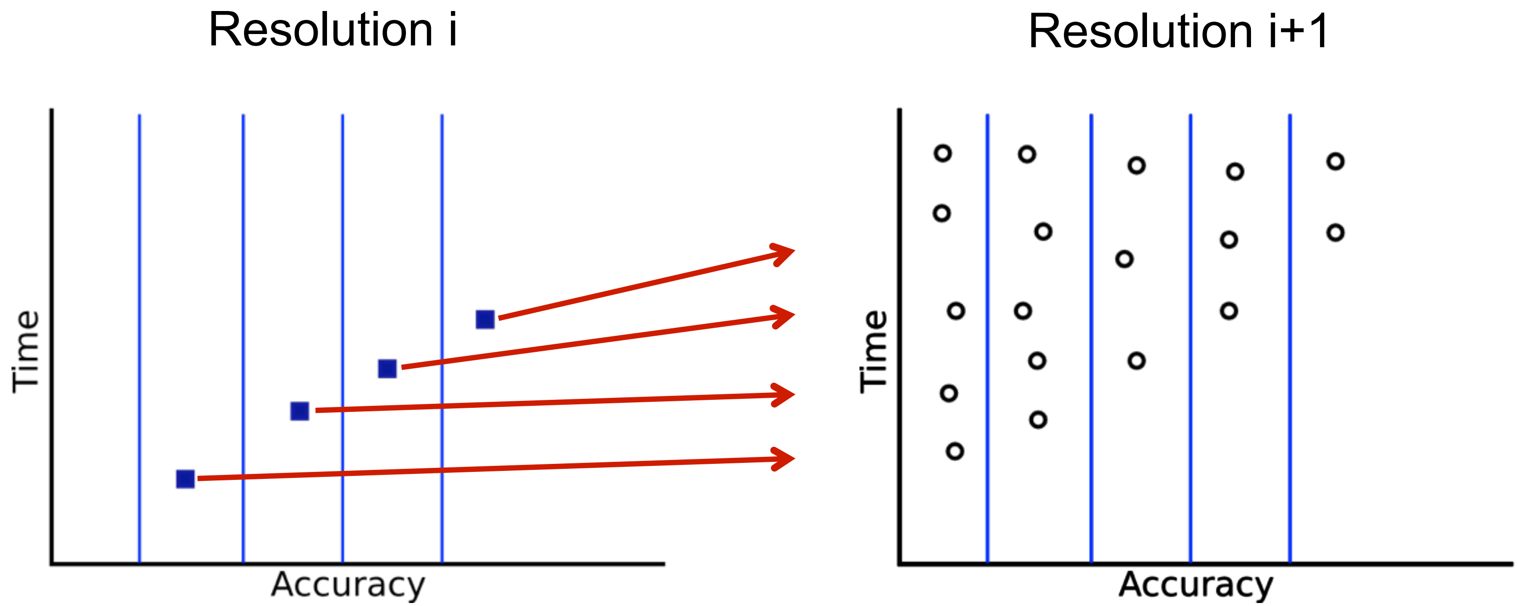


Training the Discrete Solution



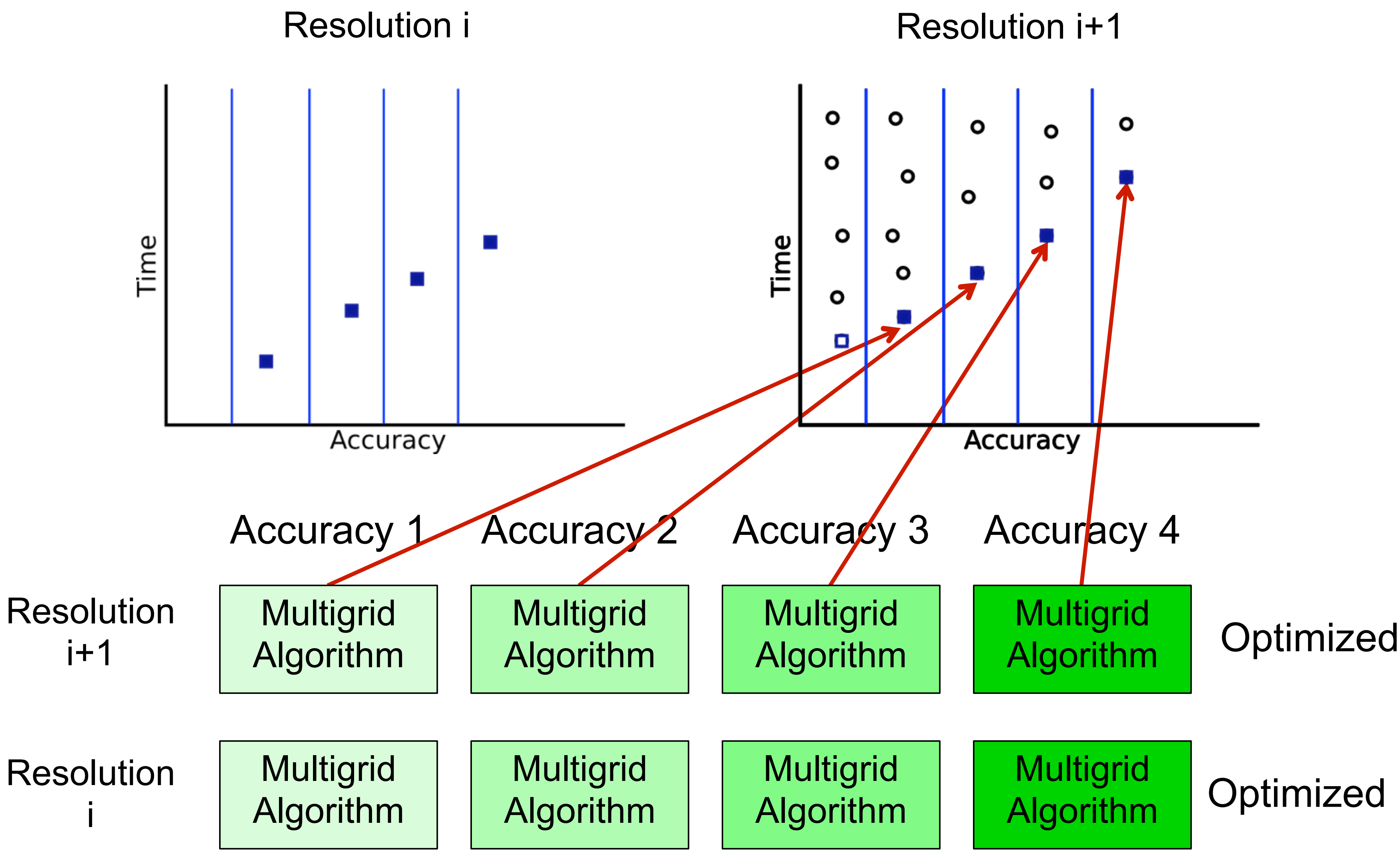
	Accuracy 1	Accuracy 2	Accuracy 3	Accuracy 4	
Resolution i+1	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Training
Resolution i	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Optimized

Training the Discrete Solution

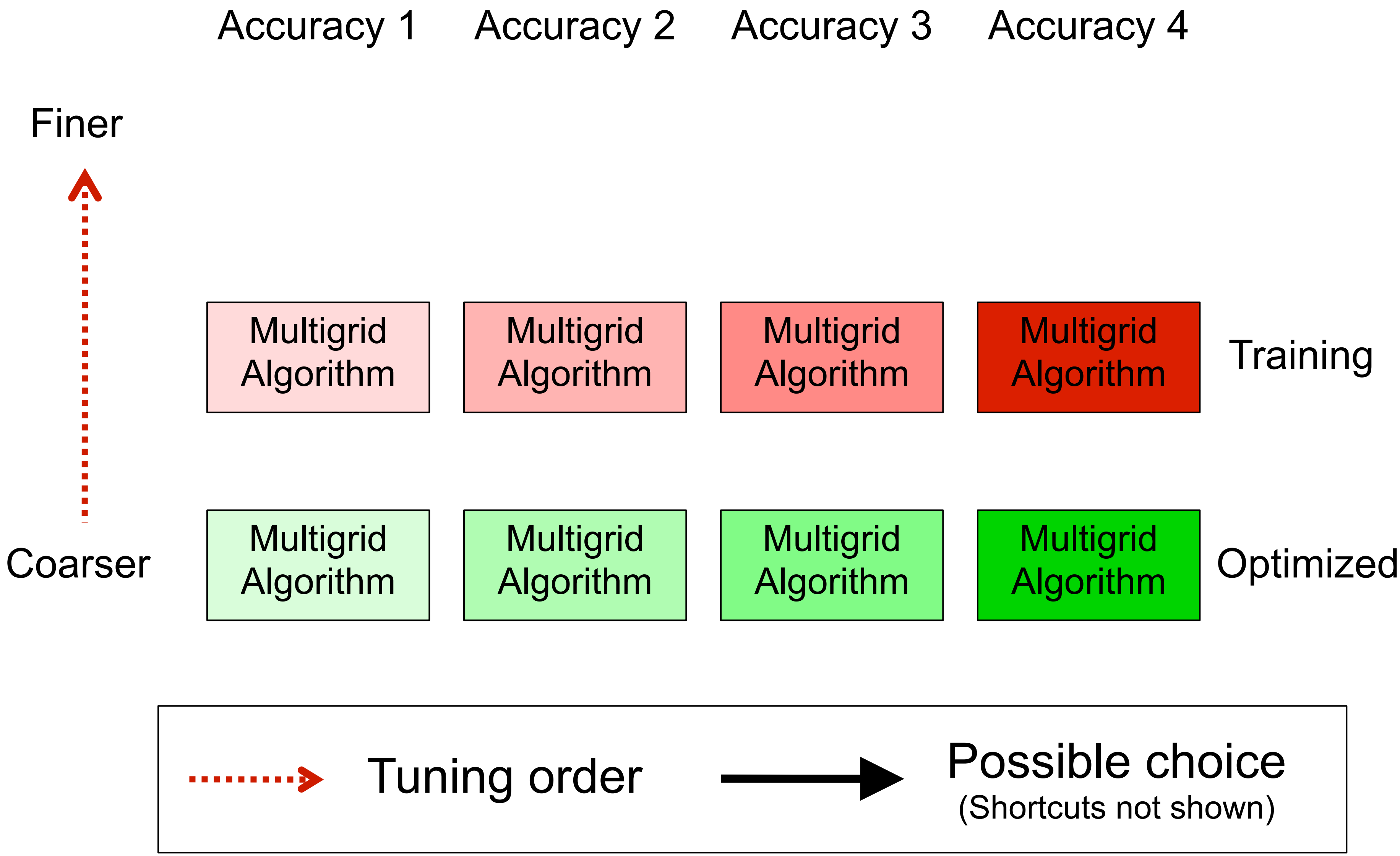


	Accuracy 1	Accuracy 2	Accuracy 3	Accuracy 4	
Resolution i+1	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Training
Resolution i	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Multigrid Algorithm	Optimized

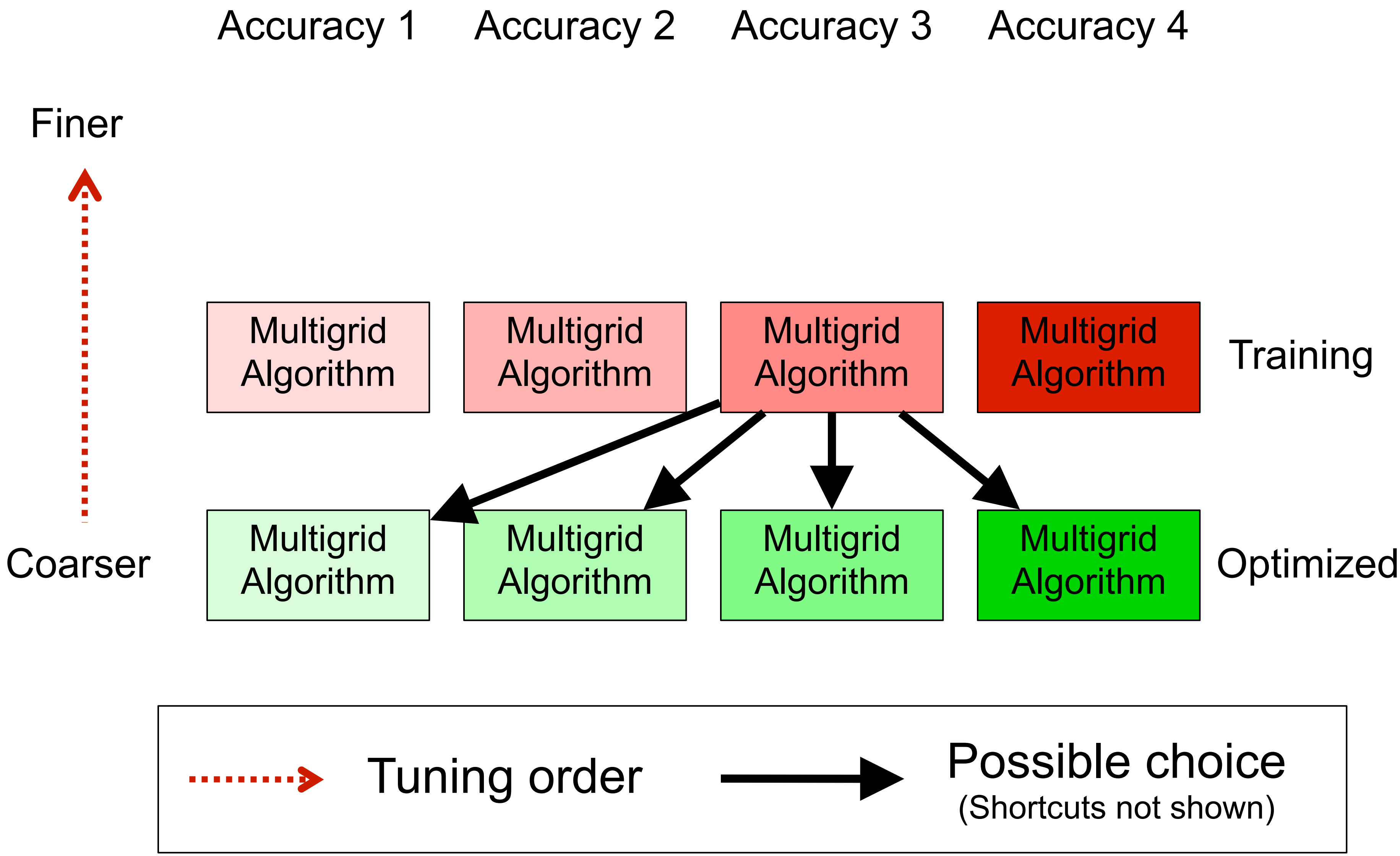
Training the Discrete Solution



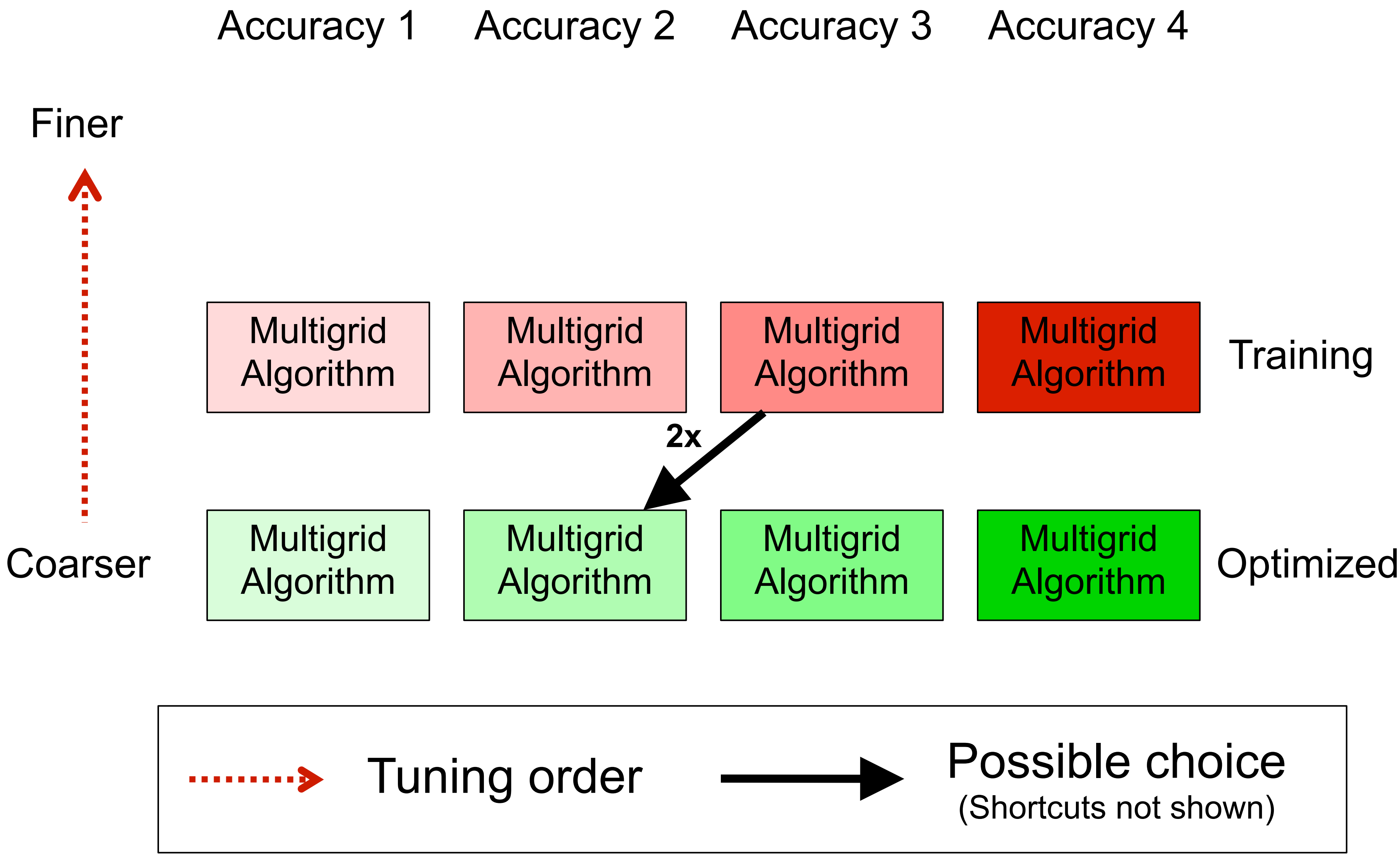
Training the Discrete Solution



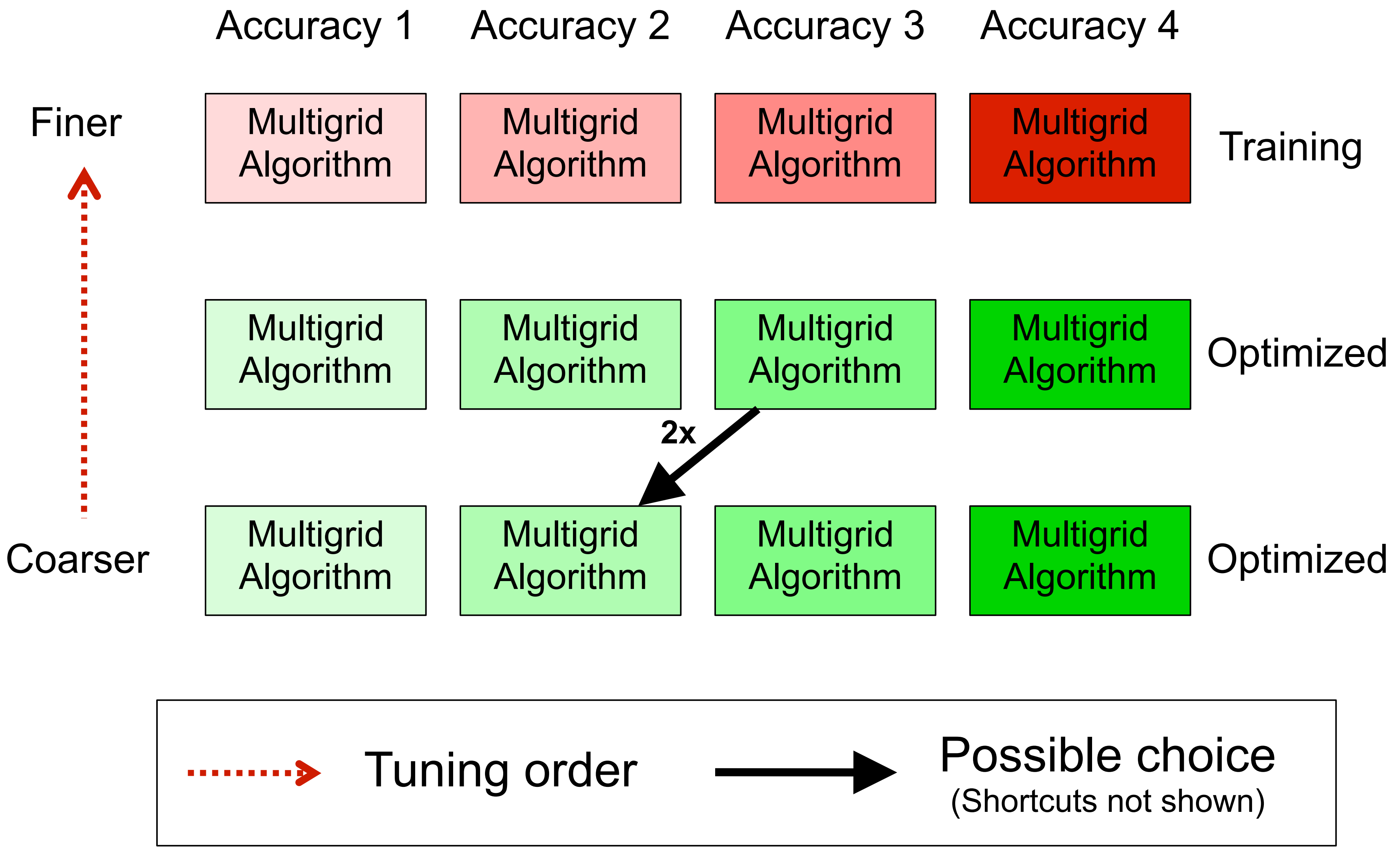
Training the Discrete Solution



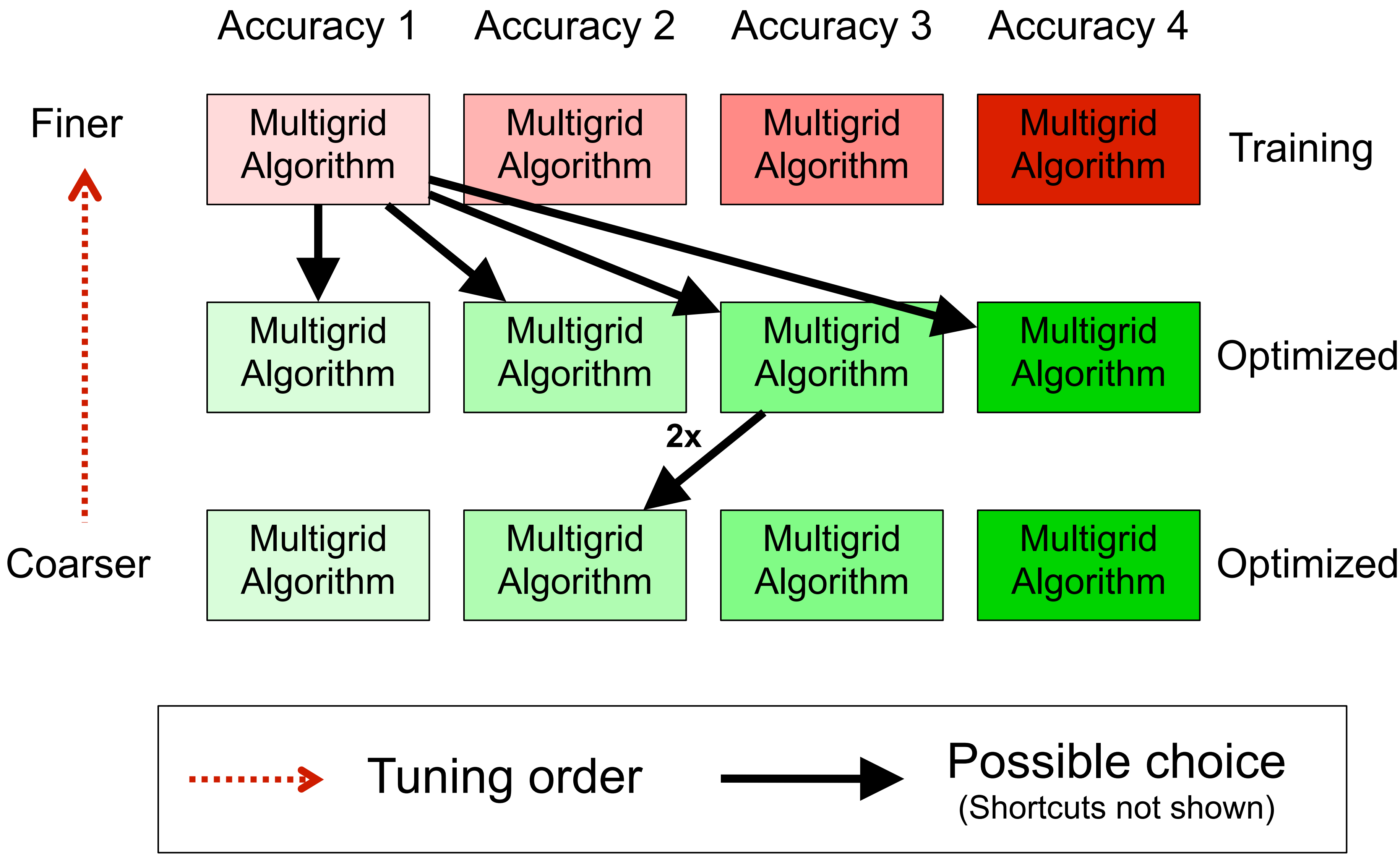
Training the Discrete Solution



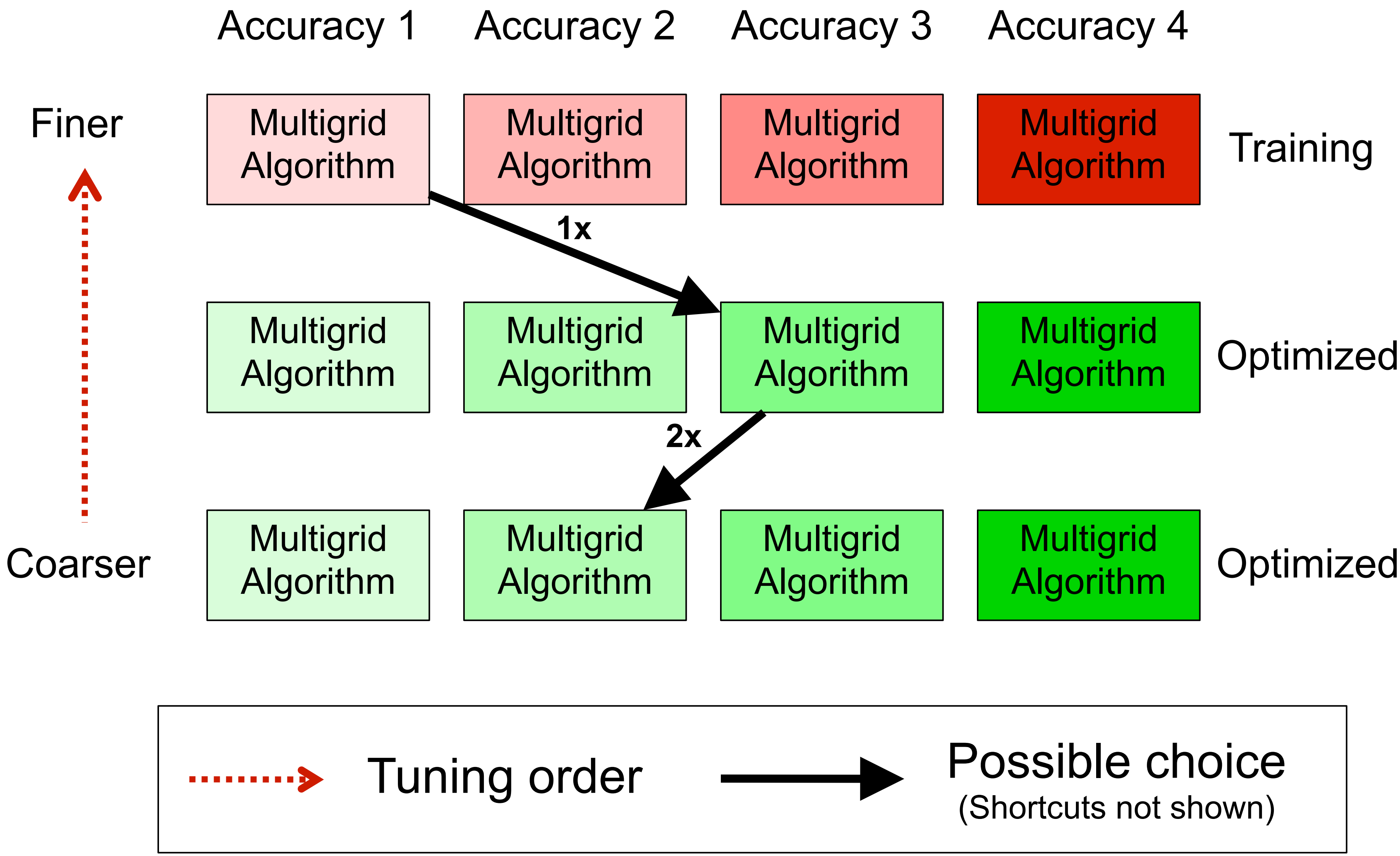
Training the Discrete Solution



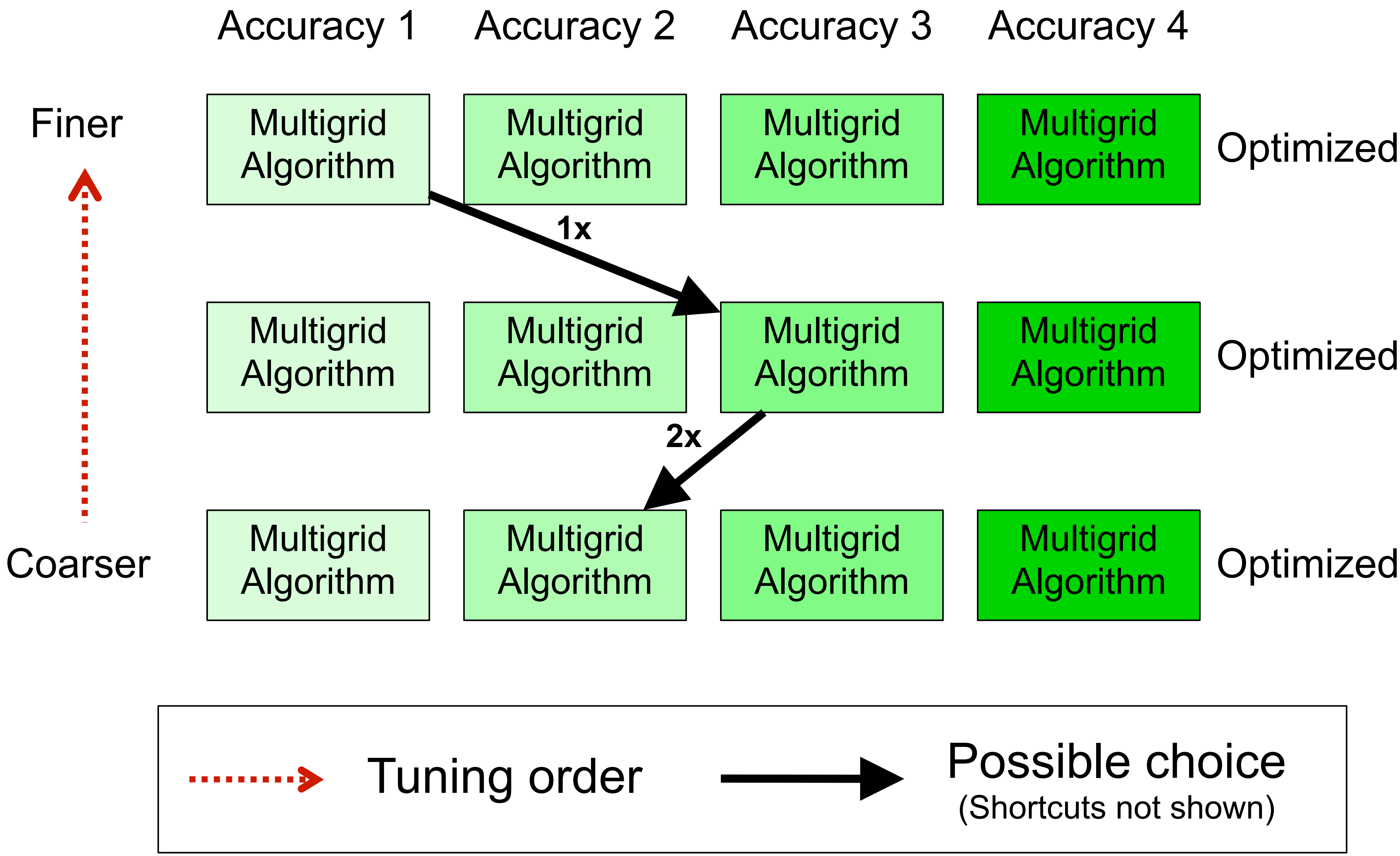
Training the Discrete Solution



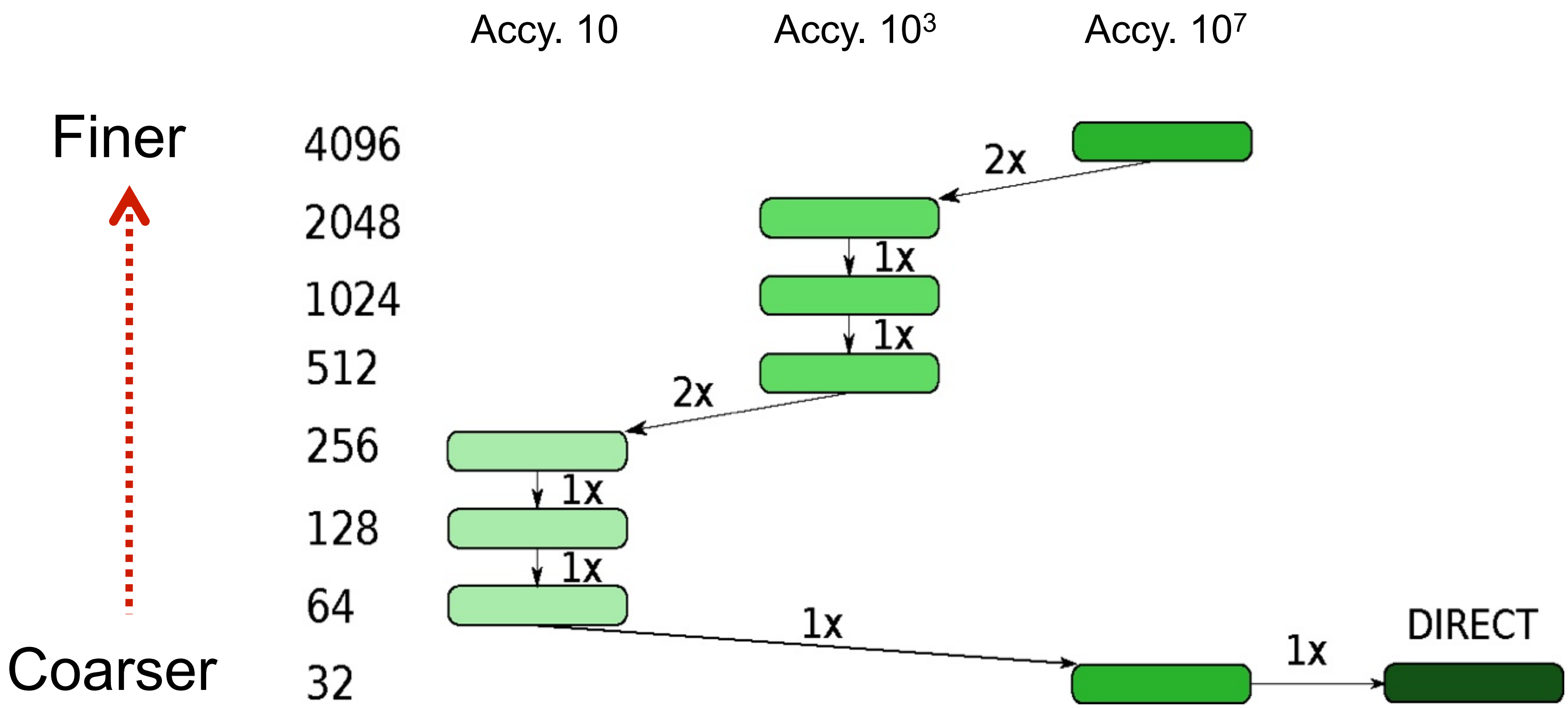
Training the Discrete Solution



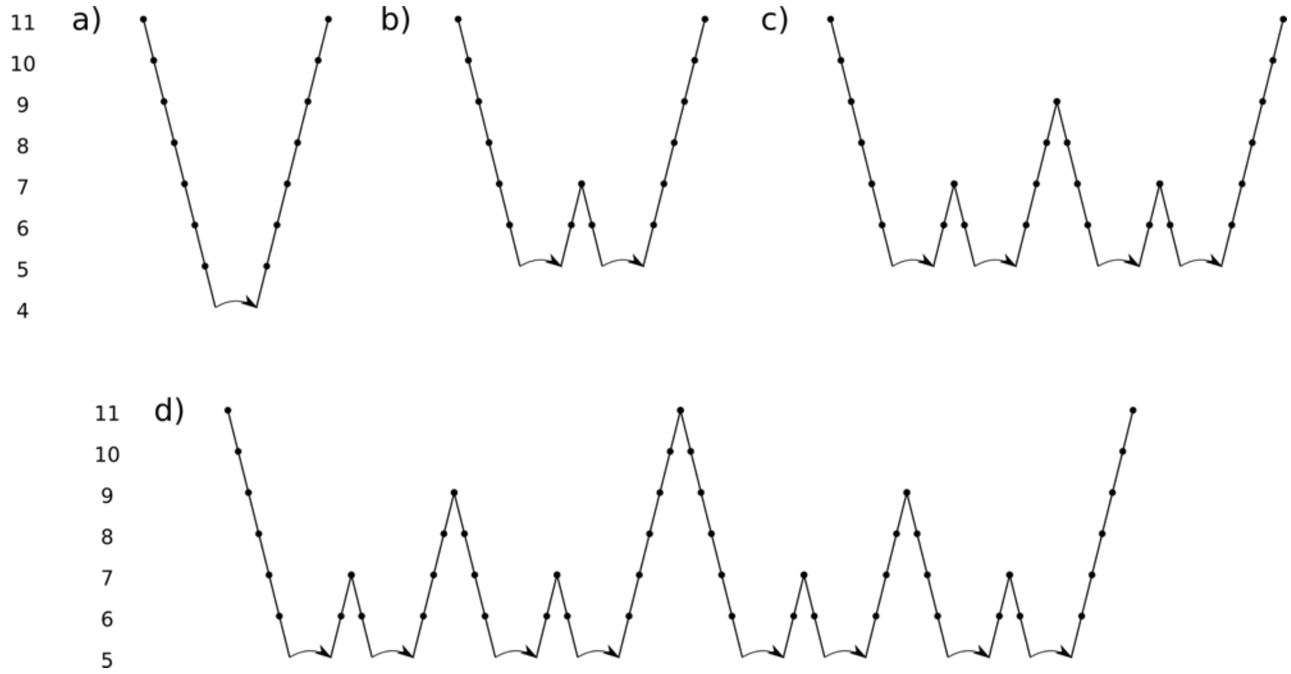
Training the Discrete Solution



Example: Auto-tuned 2D

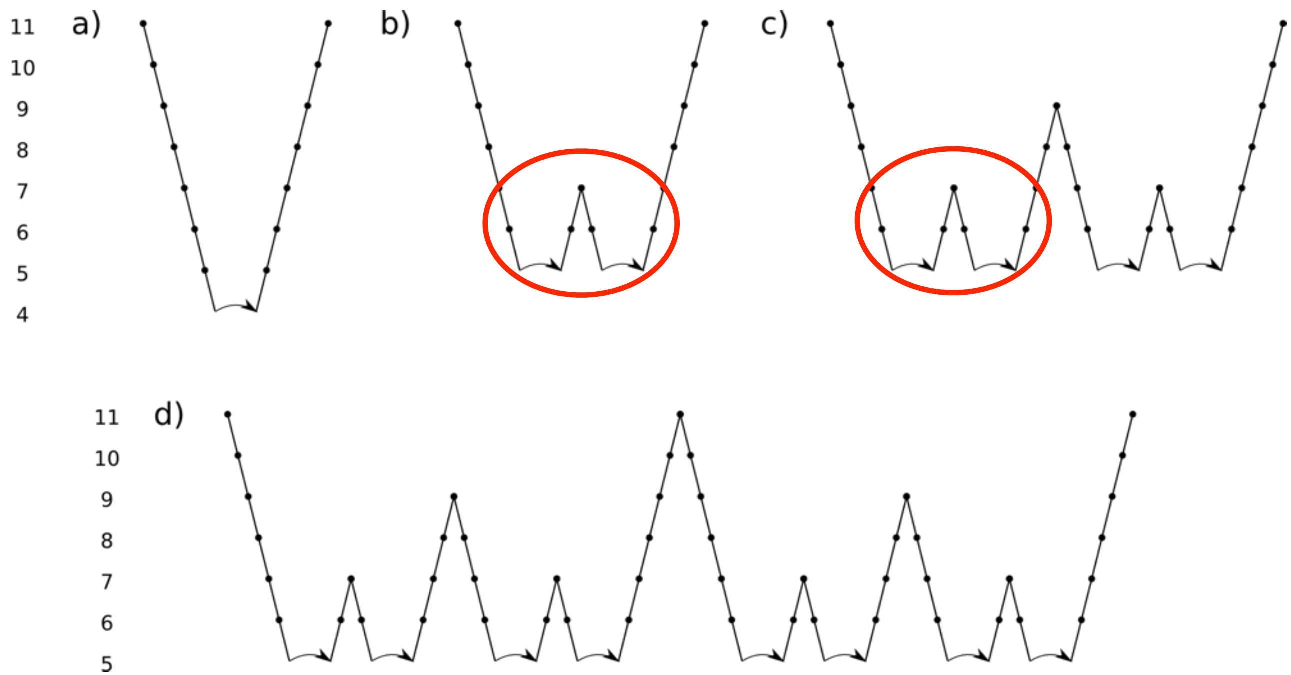


Auto-tuned Cycles for



Cycle shapes for accuracy levels a) 10, b) 10^3, c) 10^5, d) 10^7

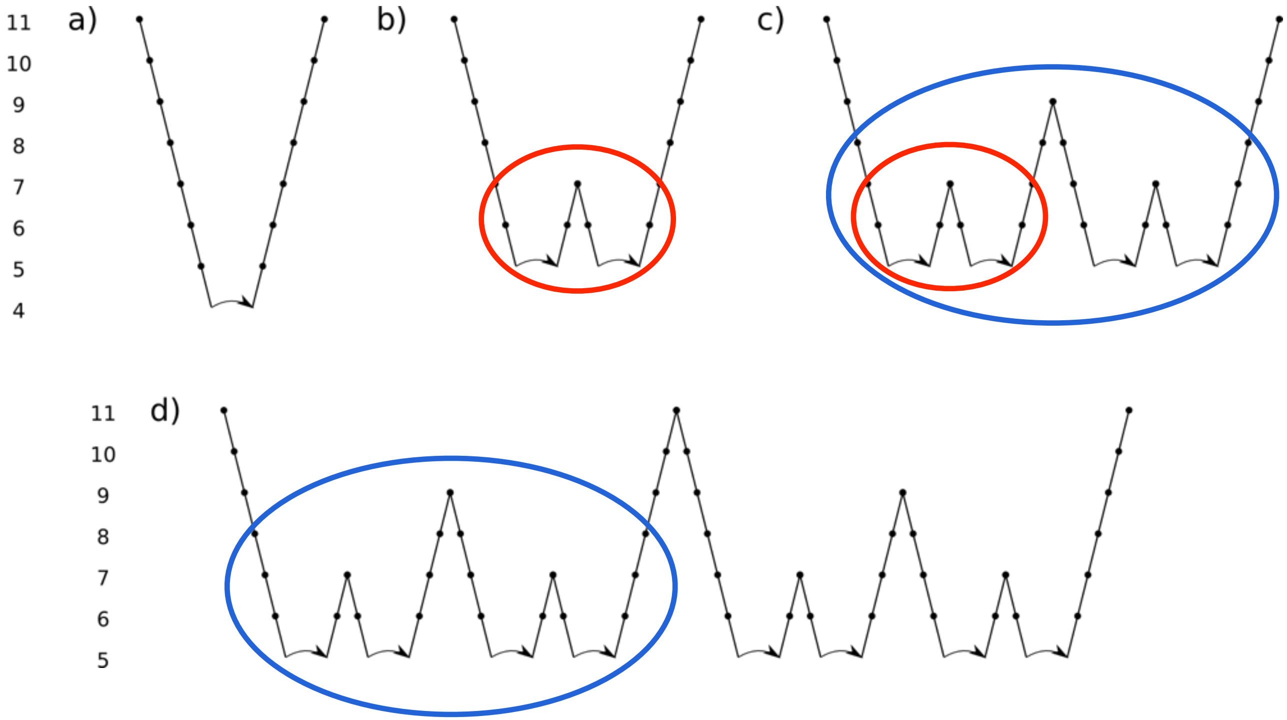
Auto-tuned Cycles for



Cycle shapes for accuracy levels a) 10, b) 10^3, c) 10^5, d) 10^7

Optimized substructures visible in cycle shapes

Auto-tuned Cycles for



Cycle shapes for accuracy levels a) 10, b) 10^3, c) 10^5, d) 10^7

Optimized substructures visible in cycle shapes

Poisson

Time

Matrix Size

Poisson

Time

Matrix Size

Binpacking – Algorithmic Choices

Data Size

Accuracy

Outline

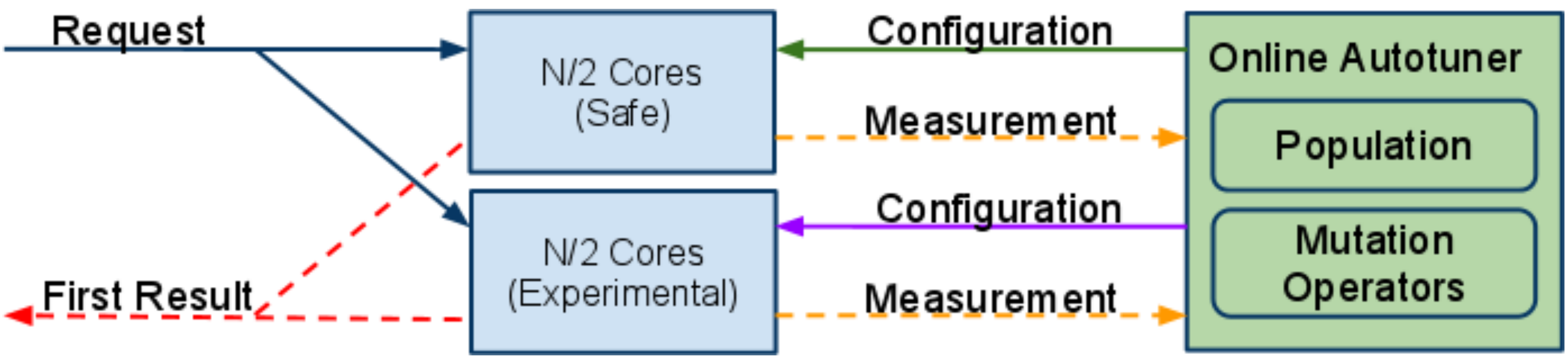
- The Three Side Stories
  - Performance and Parallelism with Multicores
  - Future Proofing Software
  - Evolution of Programming Languages
- Three Observations
- PetaBricks
  - Language
  - Compiler
  - Results
  - Variable Precision
  - Sibling Rivalry

Issues with Offline Tuning

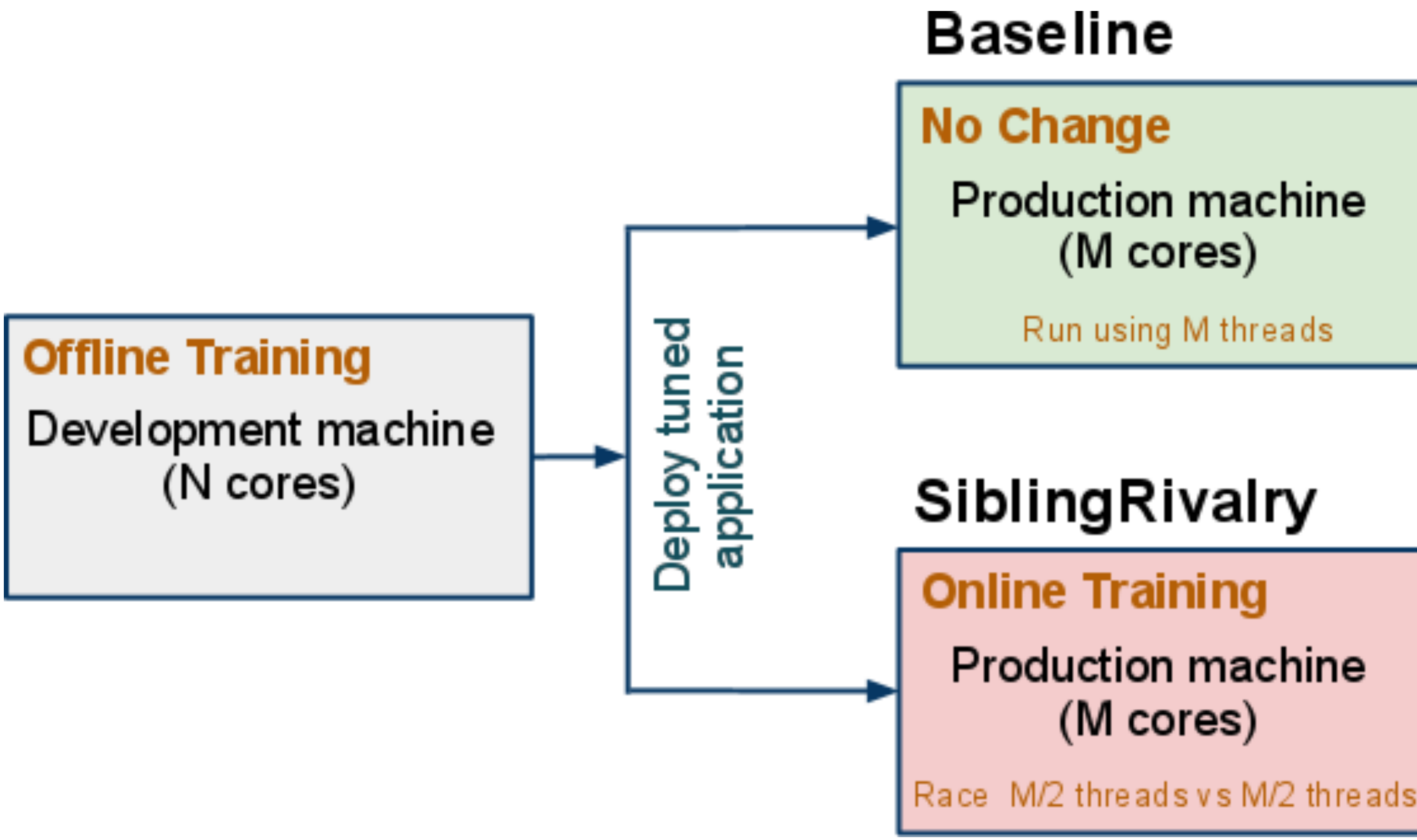
- Offline-tuning workflow burdensome
  - Programs often not re-autotuned when they should be
    - e.g. `apt-get install fftw` does not re-autotune
  - Hardware upgrades / large deployments
  - Transparent migration in the cloud
- Can't adapt to dynamic conditions
  - System load
  - Input types

SiblingRivalry: an Online Approach

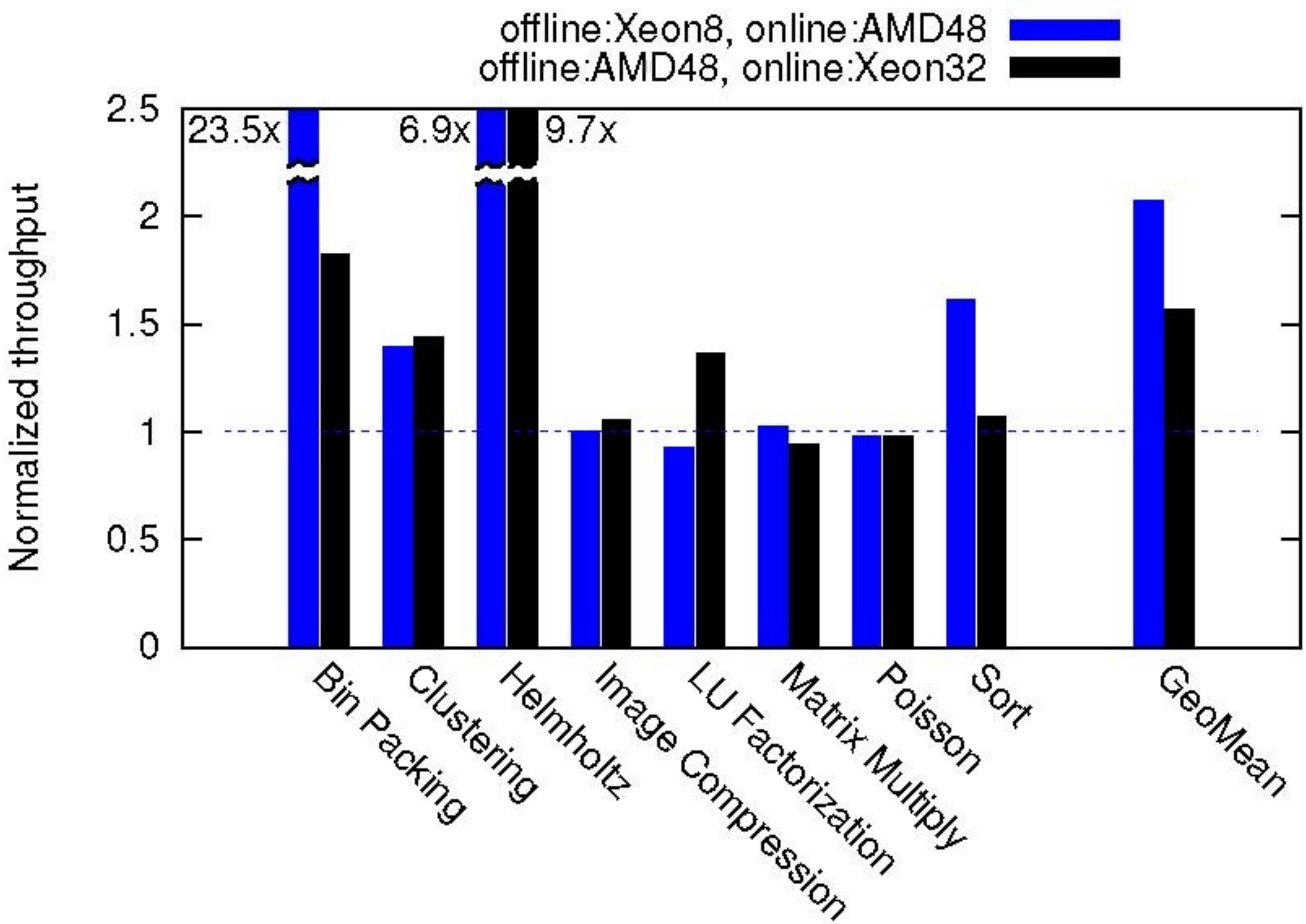
- Split available resources in half
- Process identical requests on both halves
- Race two candidate configurations (safe and experimental) and terminate slower algorithm
- Initial slowdown (from duplicating the request) can be overcome by autotuner
- Surprisingly, reduces average power consumption per request



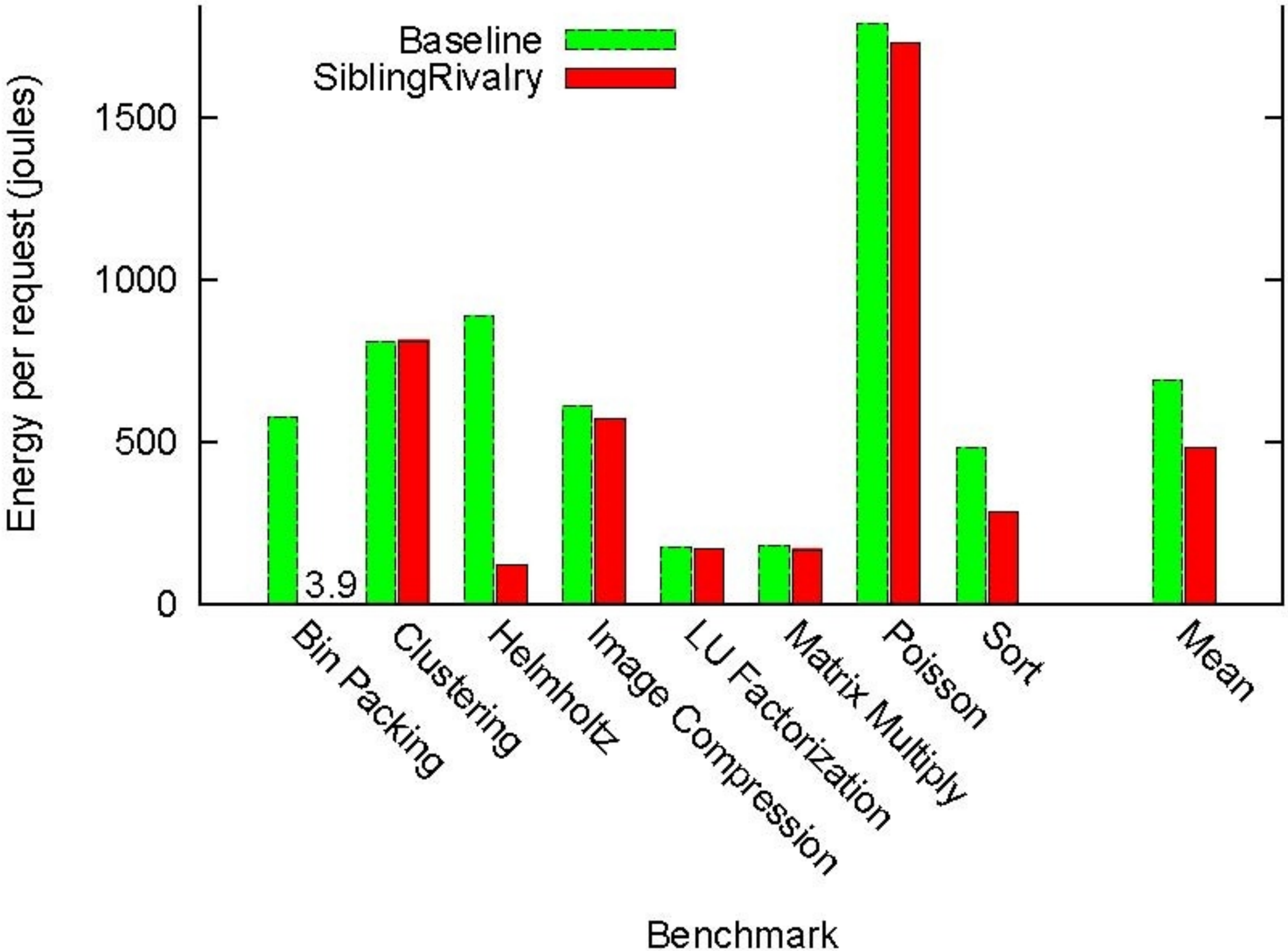
Experimental Setup



SiblingRivalry: throughput



SiblingRivalry: energy usage
(on AMD48)



Conclusion

Conclusion

- Time has come for languages based on autotuning

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)

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!

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