PetaBricks: A Language and Compiler for Algorithmic Choice

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

Presentation: Thomas Etter

Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort

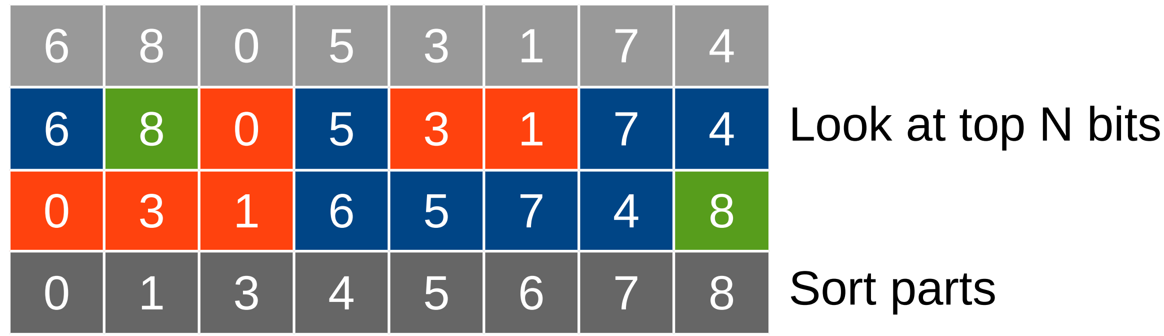
Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort

6	8	0	5	3	1	7	4	
6	8	0	5	3	1	7	4	4-way Split
6	8	0	5	1	3	4	7	Sort parts
0	1	3	4	5	6	7	8	4-way Merge

Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort



Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort

6	8	0	5	3	1	7	4	
6	8	0	5	3	1	7	4	Partition by pivot
1	3	0	5	8	6	7	4	
1	3	0	4	8	6	7	5	Swap pivot/center
0	1	3	4	5	6	7	8	Sort parts

Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort

6	8	0	5	3	1	7	4
6	8	0	5	3	1	7	4
6	8	0	5	3	1	7	4
0	6	8	5	3	1	7	4
0	5	6	8	3	1	7	4
0	3	5	6	8	1	7	4
0	1	3	5	6	8	7	4
0	1	3	5	6	7	8	4
0	1	3	4	5	6	7	8
0	1	3	4	5	6	7	8

Motivating example

- Sorting numbers
- Algorithms
  - K-way MergeSort
  - RadixSort
  - QuickSort
  - InsertionSort
- Different characteristics
- Composing the **best** hybrid sort

6	8	0	5	3	1	7	4
6	8	0	5	3	1	7	4
6	8	0	5	3	1	7	4
0	6	8	5	3	1	7	4
0	5	6	8	3	1	7	4
0	3	5	6	8	1	7	4
0	1	3	5	6	8	7	4
0	1	3	5	6	7	8	4
0	1	3	4	5	6	7	8
0	1	3	4	5	6	7	8

The Problem

- Multiple algorithms/implementations
  - Which one(s) to use?
  - In what order?
  - Cutoff points?
- For matrices:
  - Blocking size?

A New Language: Why?

- Expose algorithmic choice to the compiler
  - Parallelization
  - Automatic optimization
  - Consistency checks between choices

PetaBricks: The language

- Functional language
  - Basic construct: transform
    - Has one or more rules
  - C++ code can be directly included
    - Allows inclusion of existing libraries
  - Has facilities for dealing with matrices

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i-1) leftSum) {
        b = a + leftSum;
    }
}
```

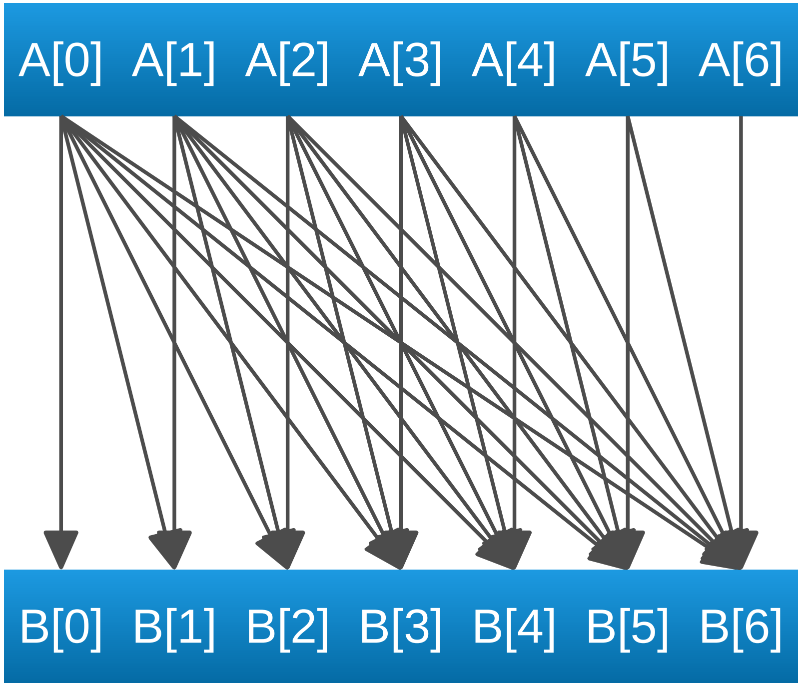
PetaBricks: The language

- RollingSum
- [1,2,3, 4, 5, 6]=> [1,3,6,10,15,21]

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i-1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: The language

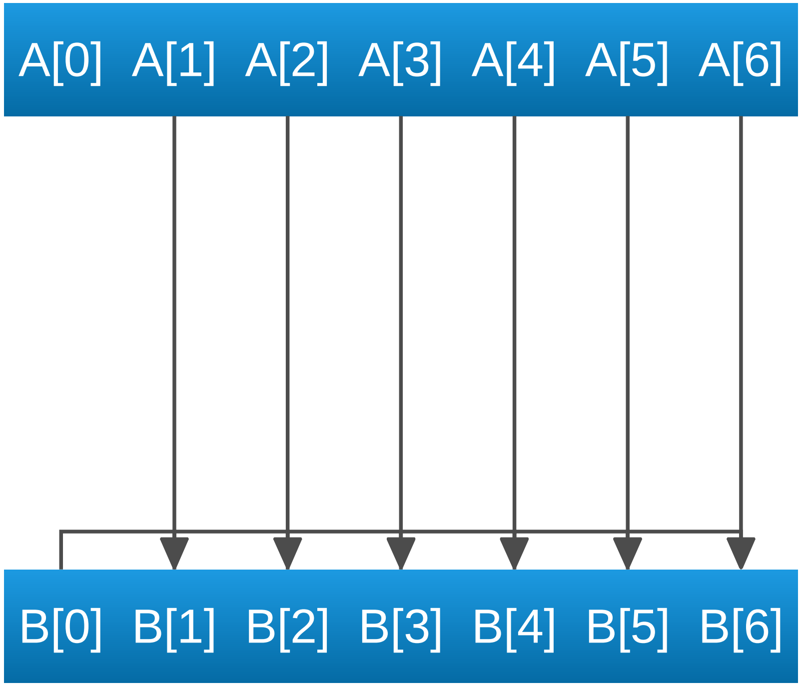
- RollingSum
- [1,2,3, 4, 5, 6]=> [1,3,6,10,15,21]
- Rule 0: $O(n^2)$




```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: The language

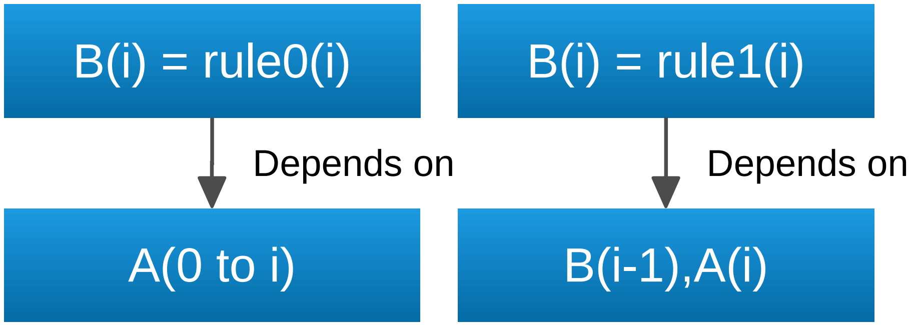
- RollingSum
- [1,2,3, 4, 5, 6]=> [1,3,6,10,15,21]
- Rule 1: O(n)




```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i-1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: Compilation

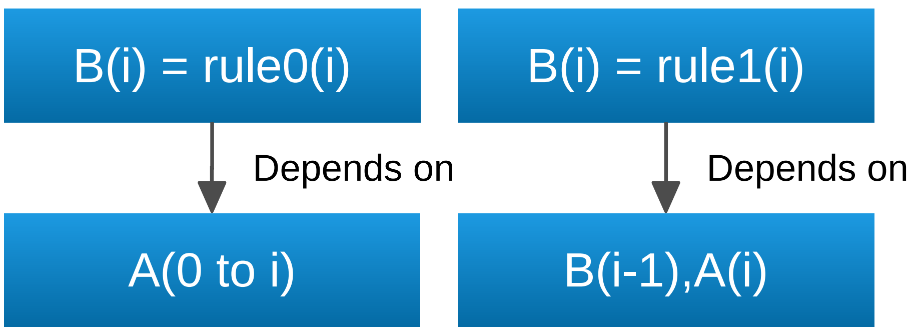
- Analyse dependencies




```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: Compilation

- Analyse dependencies

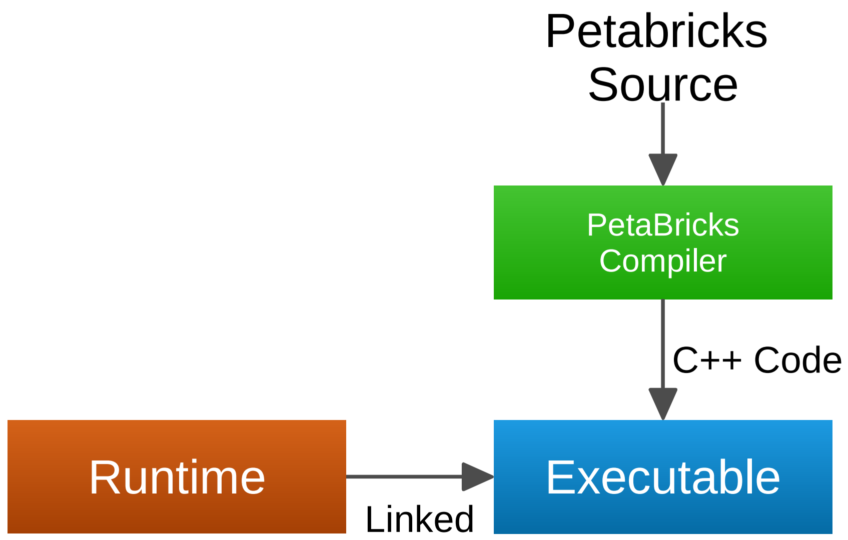


- Compute applicable regions:
  - Rule 1: [0, n)
  - Rule 2: [1, n)

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

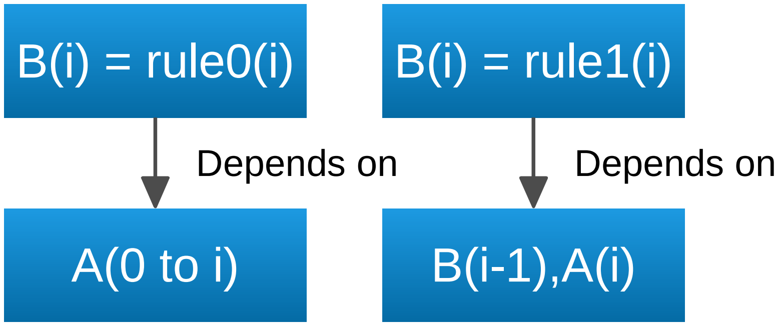
PetaBricks: The implementation

- Source-to-source compiler
  - Translates PetaBricks to C++
  - Compiles code for tuning
- Autotuning system
- Runtime library



PetaBricks: Compilation

- Analyse dependencies

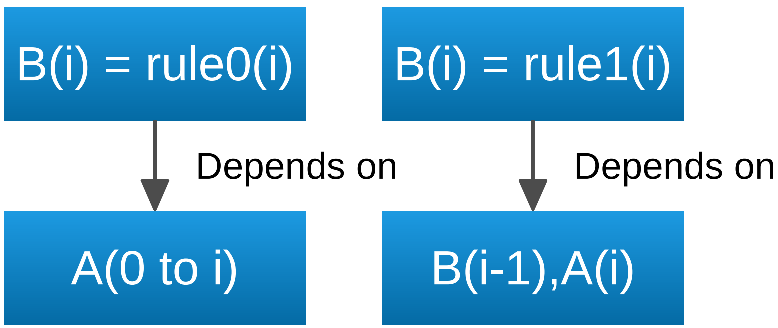


- Compute applicable regions:
  - Rule 0: [0, n)
  - Rule 1: [1, n)
- Tunable parameter: splitsize

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: Compilation

- Analyse dependencies

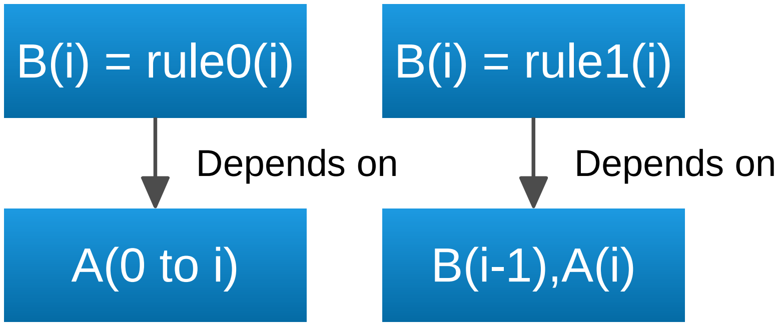


- Compute applicable regions:
  - Rule 0: [0, n)
  - Rule 1: [1, n)
- Tunable parameter: splitsize

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

PetaBricks: Compilation

- Analyse dependencies

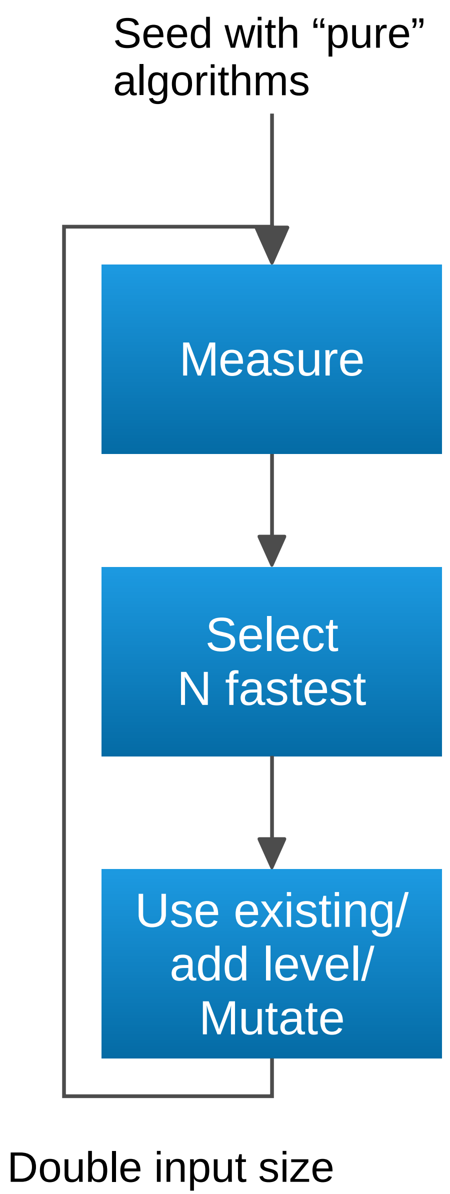


- Compute applicable regions:
  - Rule 0: [0, n)
  - Rule 1: [1, n)
- Tunable parameter: splitsize

```
transform RollingSum
from A[ n ]
to B[ n ]
{
    //rule 0: sum all elements to the left
    to ( B.cell (i) b )
    from (A.region (0, i) in ) {
        b=sum(in) ;
    }
    //rule 1: use the previously computed value
    to (B.cell (i) b )
    from (A.cell (i) a ,
    B.cell (i−1) leftSum) {
        b = a + leftSum;
    }
}
```

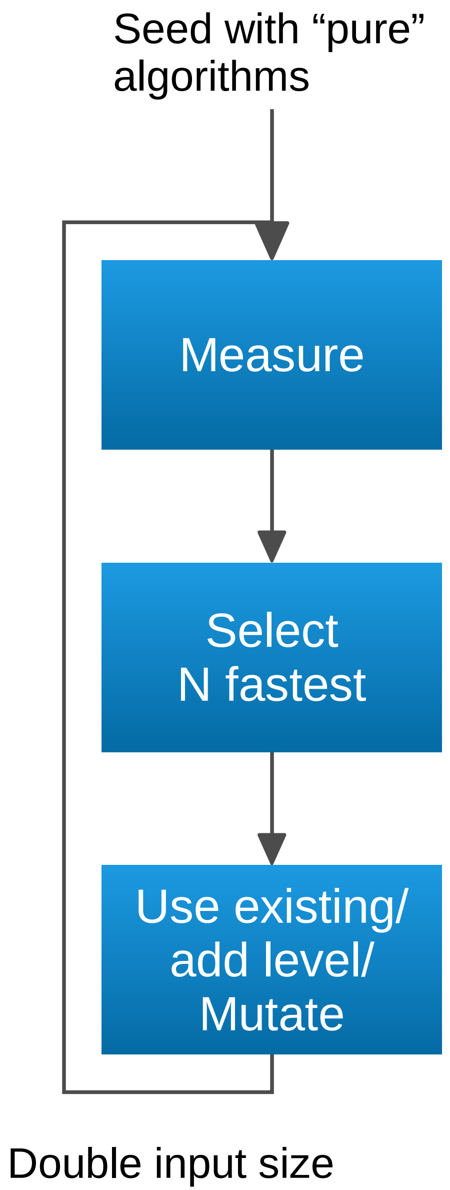
Tuning

- Tune bottom-up
  - Start small
  - Evolve configurations
- Tune additional parameters
  - Parallel-sequential cutoff points



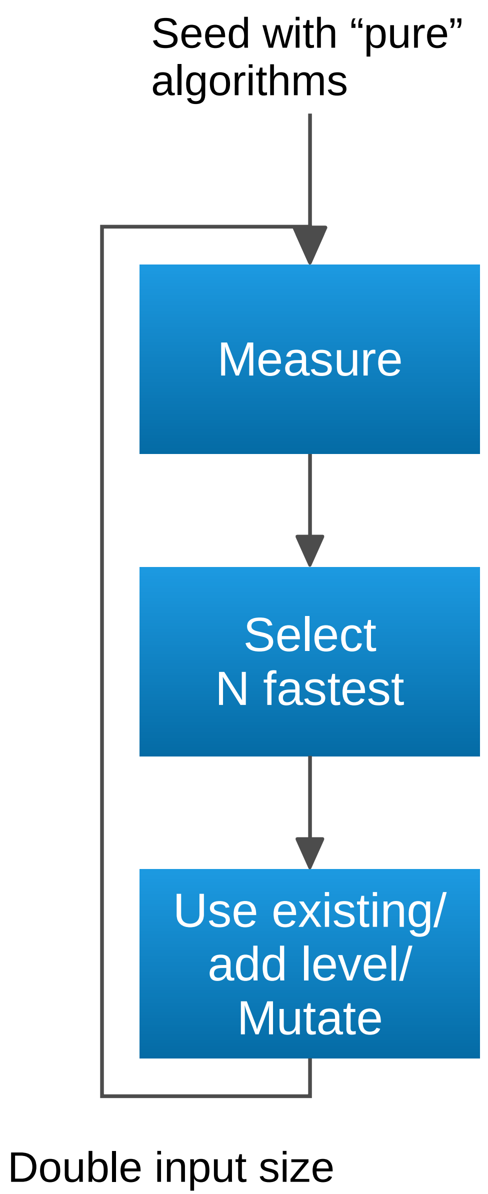
Tuning

- Tune bottom-up
  - Start small
  - Evolve configurations
- Tune additional parameters
  - Parallel-sequential cutoff points



Tuning

- Tune bottom-up
  - Start
    - all single-algorithm implementations
    - small training input
      - Double input every iteration
  - Keep the N fastest algorithms
    - Extend/Mutate the fastest algorithms
- Tune additional parameters
  - Parallel-sequential cutoff points



Automatic Blocking

- AB[w,h] = A[c,h] * B[w,c]

```
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));
    }
}
```

Automatic Blocking

- AB[w,h] = A[c,h] * B[w,c]

```
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));
    }
}
```

Automatic Blocking

- AB[w,h] = A[c,h] * B[w,c]
  - Blocking on c is non-trivial

```
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));
    }
}
```

Automatic Blocking

```
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));
    }
}
```

Automatic Blocking

- Dependency analysis yields:

```
// 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);
    }
```

```
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));
    }
}
```

Automatic Blocking

- Dependency analysis yields:

```
// 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);
    }
```

```
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));
    }
}
```

Automatic Blocking

- Dependency analysis yields:

```
// 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);
    }
```

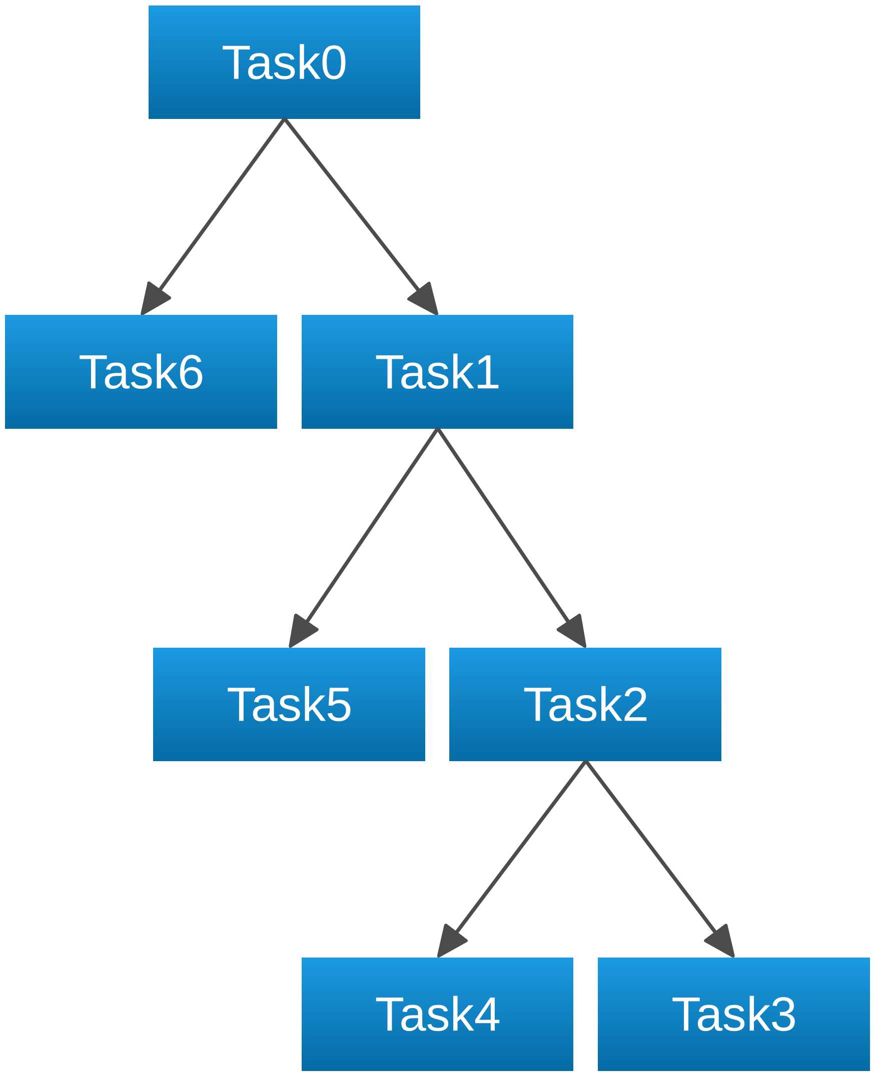
```
transform MatrixMultiply
 from A[c,h], B[w,c]
 to AB[w,h]
 {
    // Base case, compute a single element
    to(AB.cell(x,y) out)
    from(A.row(y) a, B.column(x) b) {
        out = dot(a,b);
    }
    //  Recursively decompose in c
    to(AB ab)
    from(A.region( 0, 0, c/2,   h) a1,
        A.region(c/2, 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));
    }
}
```

Runtime Library

- Allows to pass configuration
- Handles Multithreading
  - Task queue for every thread
  - Task-stealing protocol for other threads

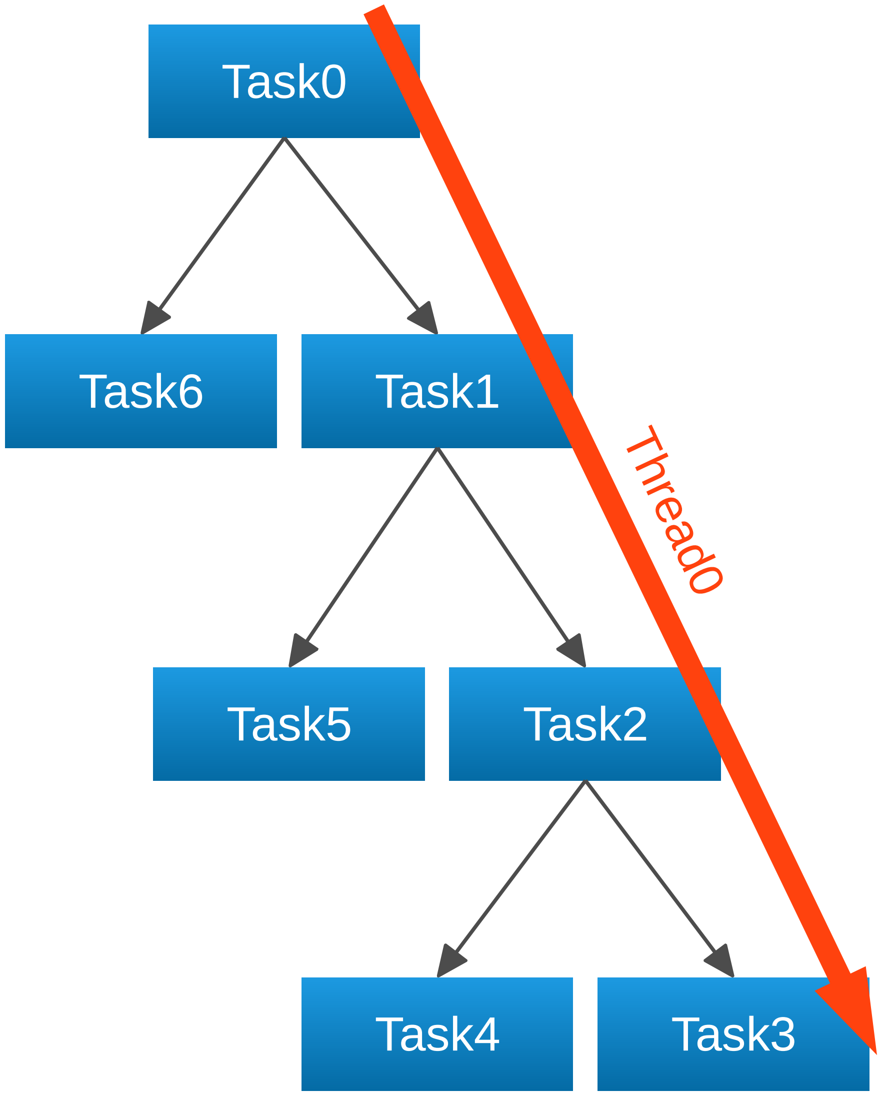
Task-stealing runtime

- DFS (depth first search)
- Execution order with one thread



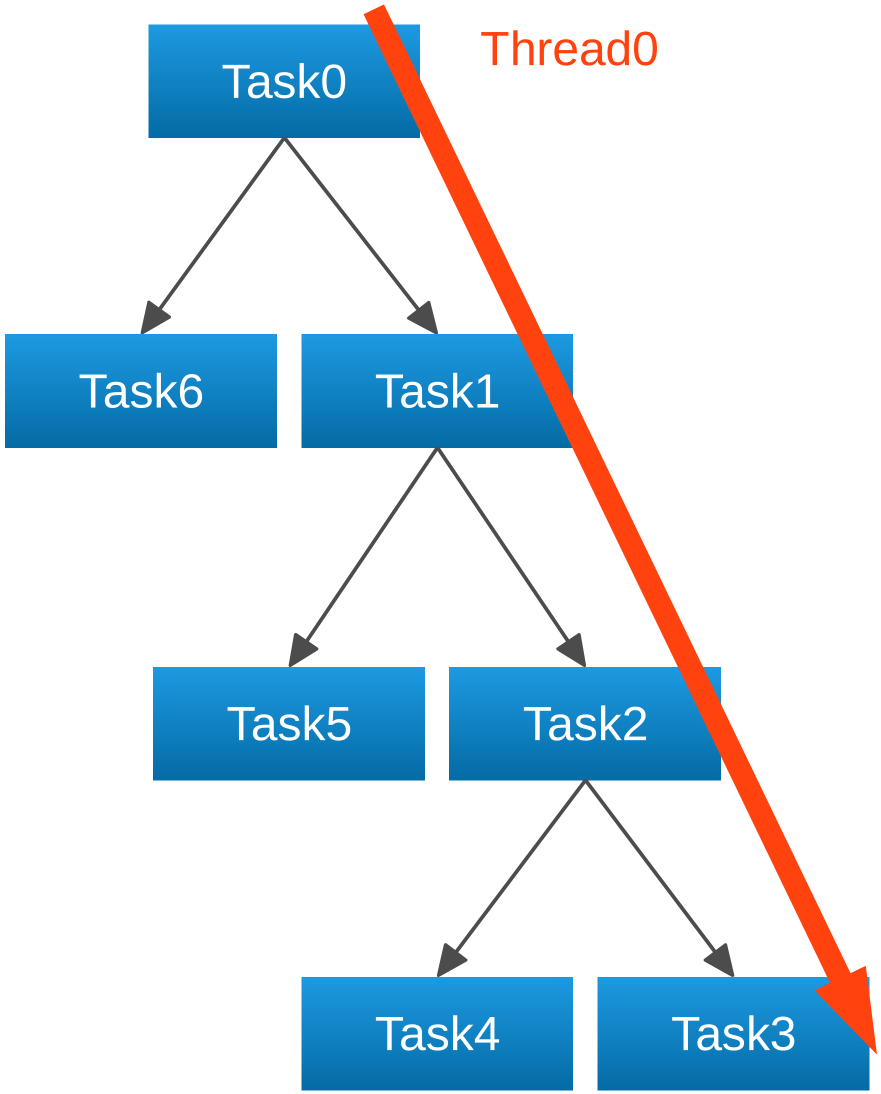
Thread-stealing runtime

- We hold Thread1
  - The Queue will be:



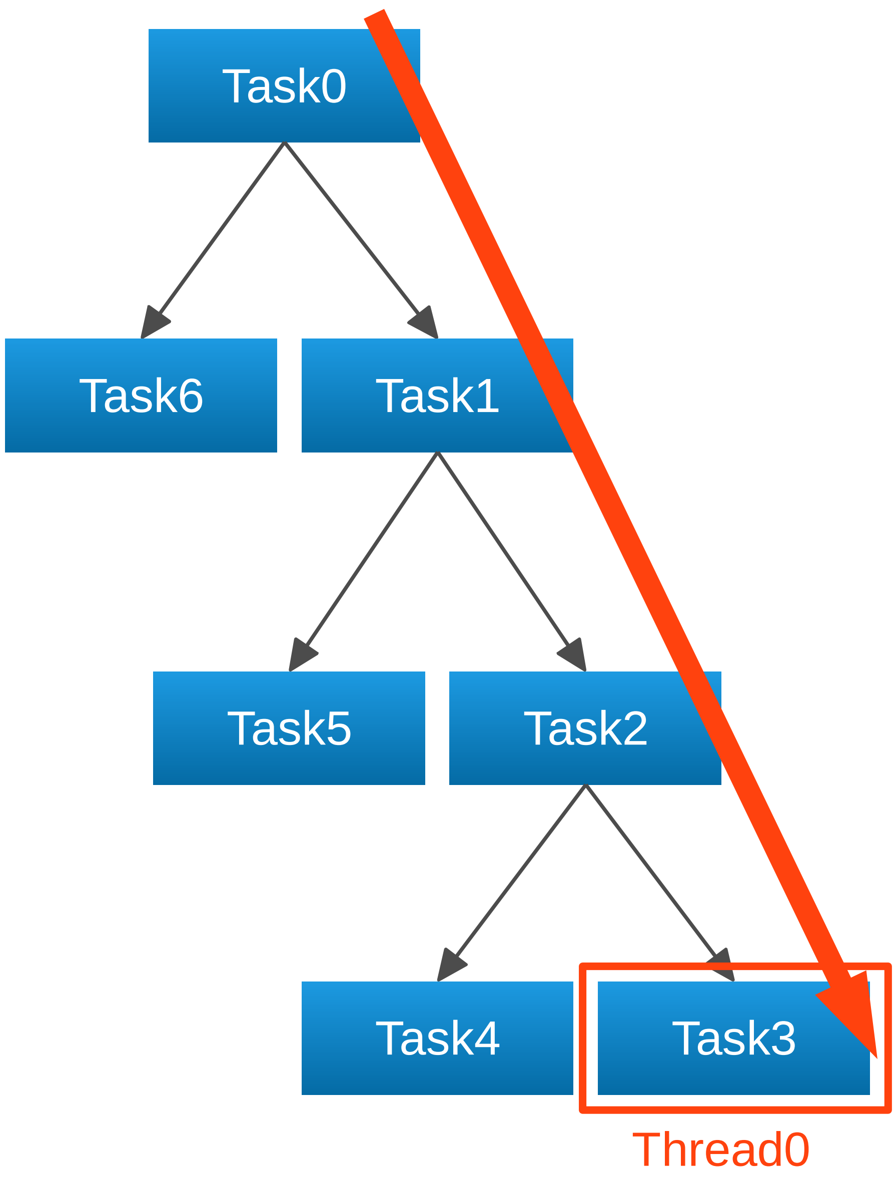
Task-stealing example

- We pause Thread1
  - Thread0 runs till Task3



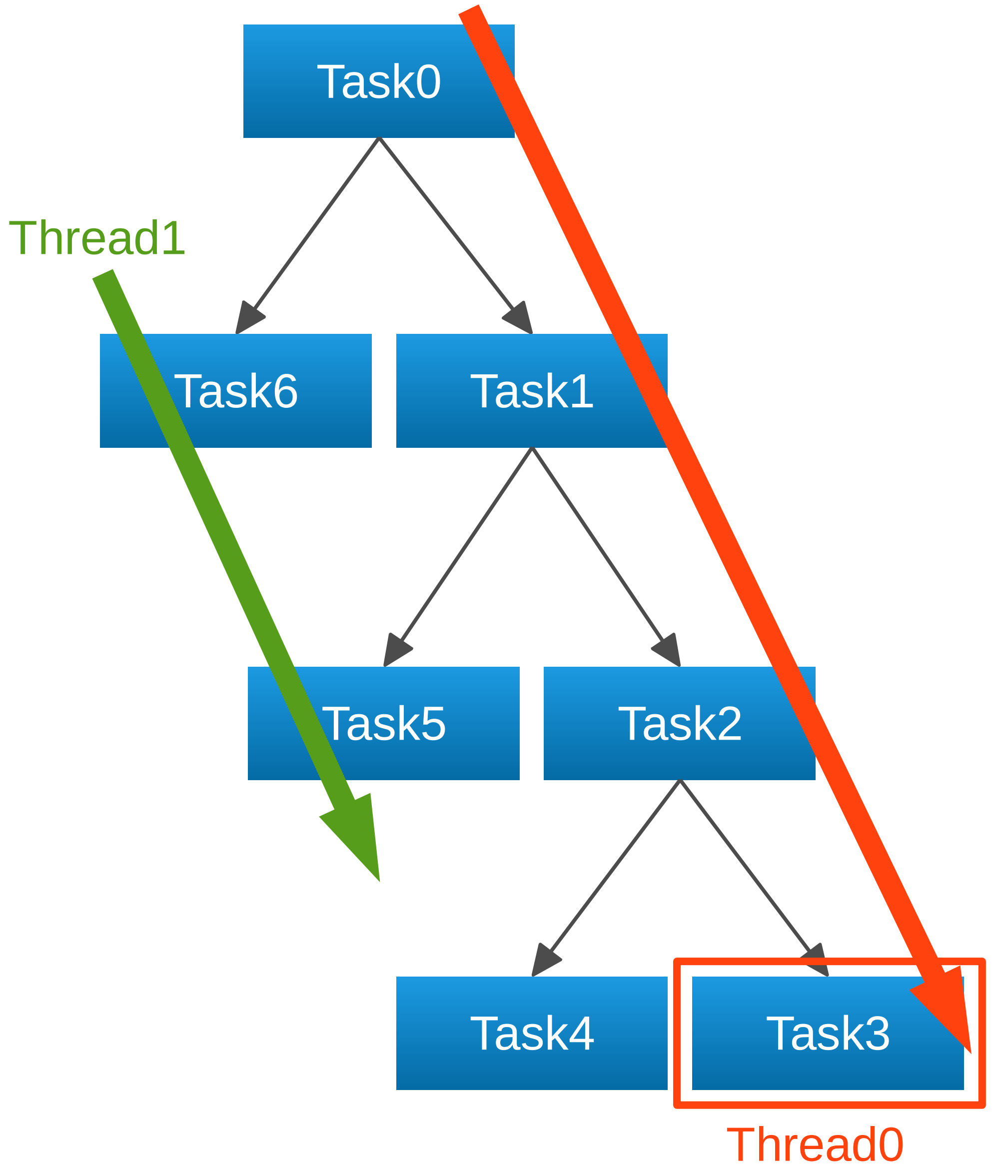
Task-stealing example

- We pause Thread1
  - Thread0 runs till Task3
  - The Queue is now:
    6 5 4



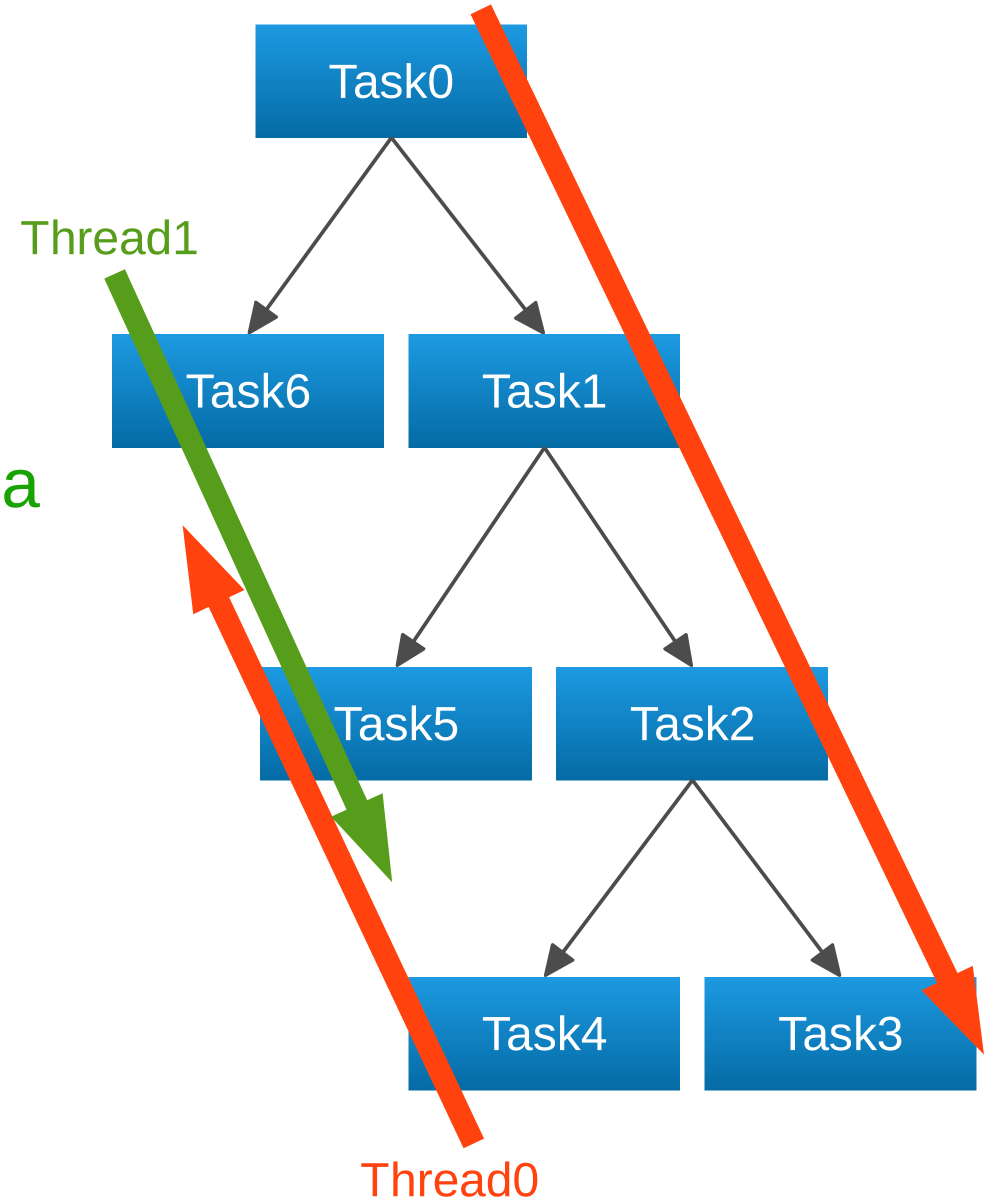
Task-stealing example

- The queue is:
  6 5 4
- resume Thread1
- Thread1 steals 6 from Thread0's queue



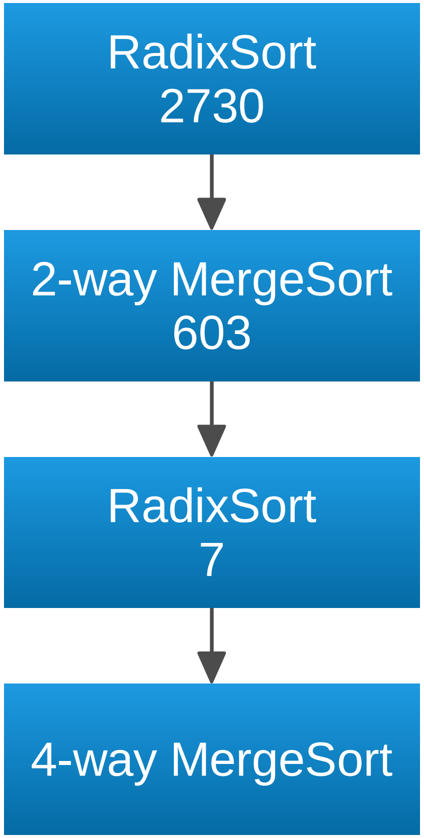
Task-stealing example

- The queue is:
  6 5 4
- resume Thread1
- Thread1 steals 6 from Thread0's queue
- Thread0 uses it's queue as a stack → continues at 4



Output

- Optimized for 2^{20} doubles on a Core 2 Duo with 2 threads:
  - Sequential cutoff: 774



Results

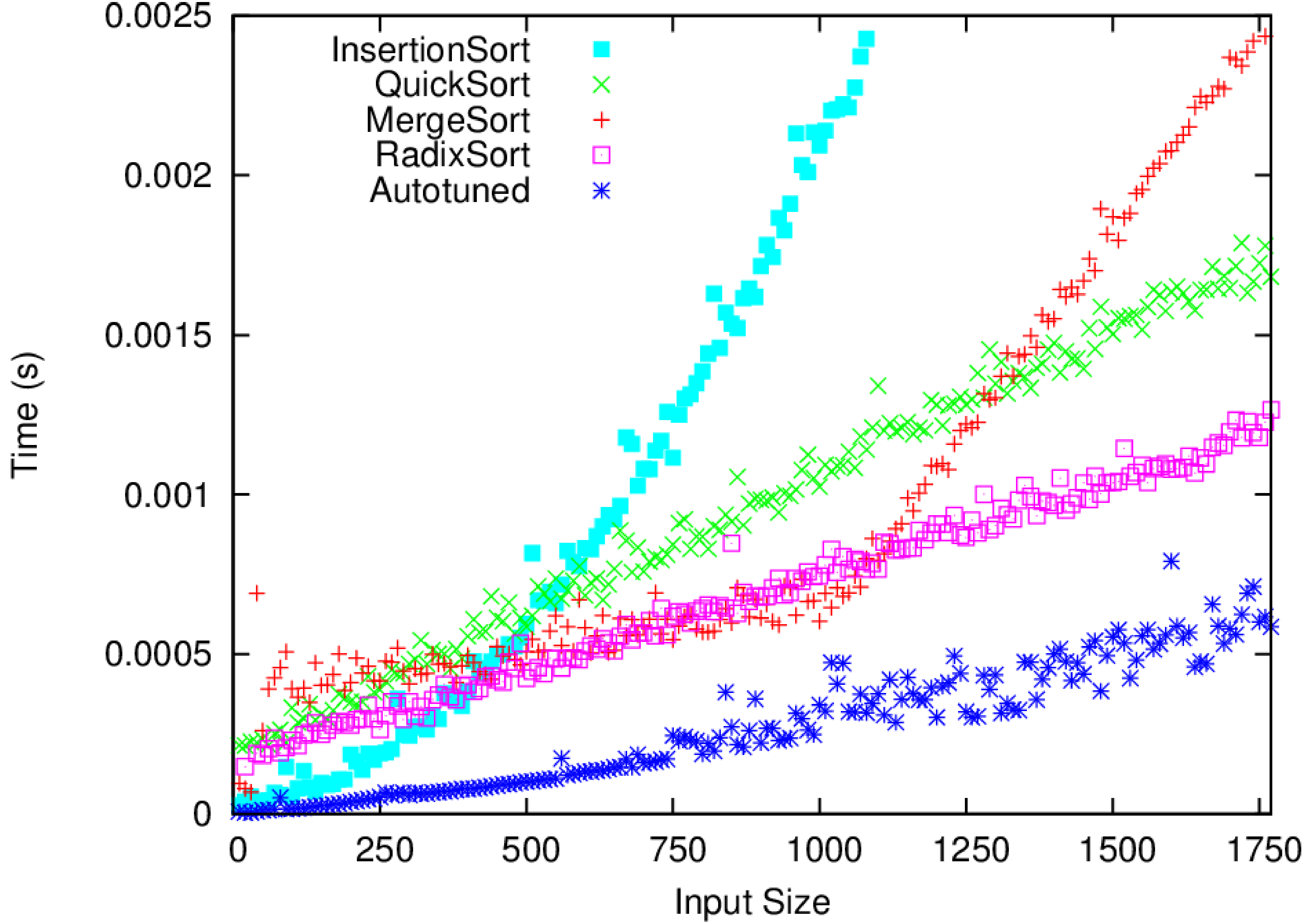


Figure 14. Performance for sort on 8 cores.

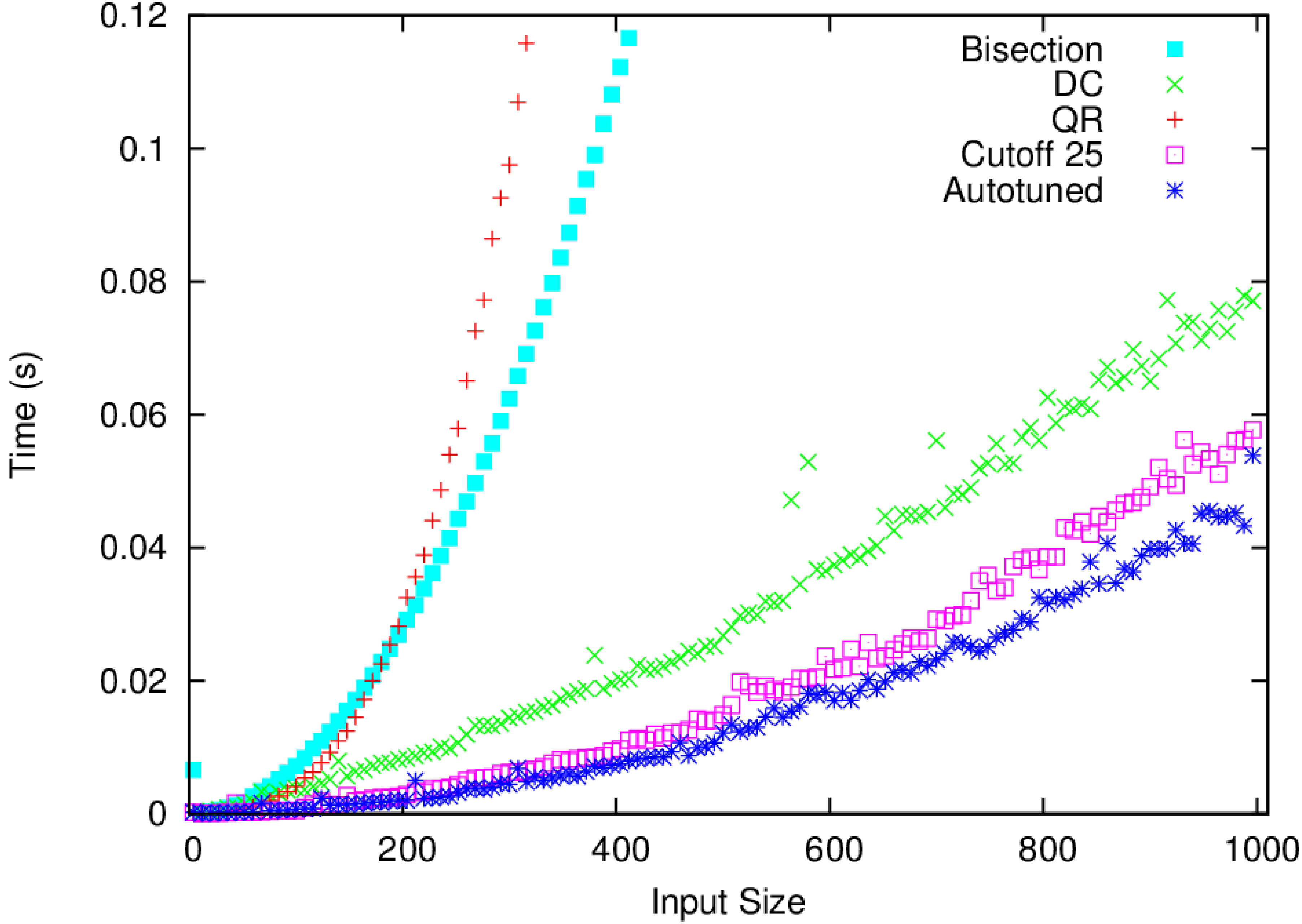


Figure 12. Performance for Eigenproblem on 8 cores. "Cutoff 25" corresponds to the hard-coded hybrid algorithm found in LAPACK.

Source: Paper

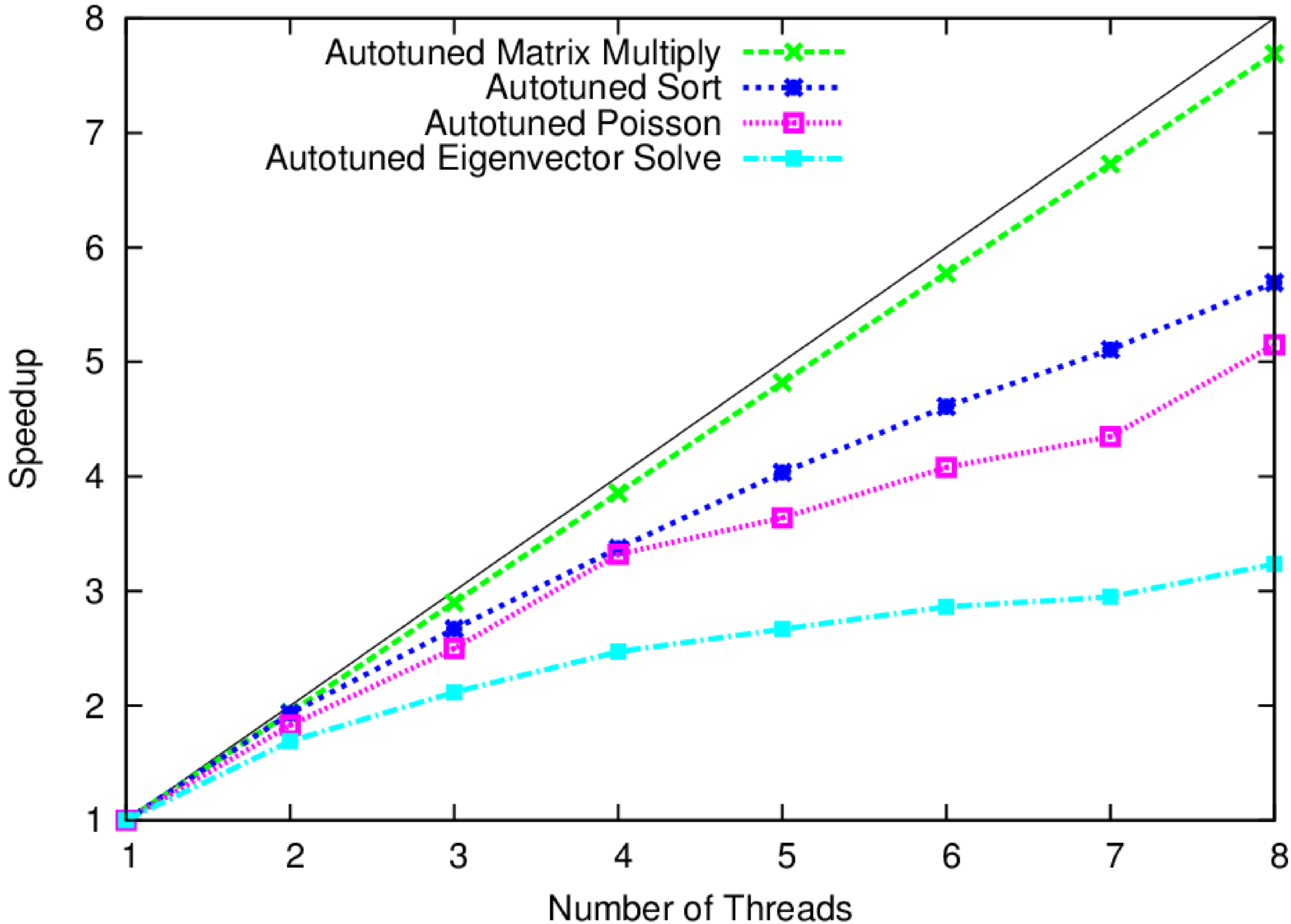


Figure 16. Parallel scalability. Speedup as more worker threads are added. Run on an 8-way (2 processor × 4 core) x86_64 Intel Xeon System.

Source: Paper

Run on		Trained on			
		Mobile	Xeon 1-way	Xeon 8-way	Niagara
	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	-

Source: Paper

Conclusion

- It is possible to have choice embedded in a programming language
- Pro
  - Good Performance (can beat LAPACK)
  - Easy adaption to different core counts
  - Numbers can be extracted
  - Free software
- Contra
  - New language
  - Overhead
  - Parts written in Python

Conclusion

- It is possible to have choice embedded in a programming language
- Pro
  - Good Performance (can beat LAPACK)
  - Easy adaption to different core counts
  - Numbers can be extracted
  - Free software
- Contra
  - New language
  - Overhead
  - Parts written in Python

Conclusion

- It is possible to have choice embedded in a programming language
- Pro
  - Good Performance (can beat LAPACK)
  - Easy adaption to different core counts
  - Numbers can be extracted
  - Free software
- Contra
  - New language
  - Overhead
  - Parts written in Python

Questions?

Abbreviation	System	Frequency	Cores used	Scalability	Algorithm Choices (w/ switching points)
Mobile	Core 2 Duo Mobile	1.6 GHz	2 of 2	1.92	IS(150) 8MS(600) 4MS(1295) 2MS(38400) QS(∞)
Xeon 1-way	Xeon E7340 (2 x 4 core)	2.4 GHz	1 of 8	-	IS(75) 4MS(98) RS(∞)
Xeon 8-way	Xeon E7340 (2 x 4 core)	2.4 GHz	8 of 8	5.69	IS(600) QS(1420) 2MS(∞)
Niagara	Sun Fire T200 Niagara	1.2 GHz	8 of 8	7.79	16MS(75) 8MS(1461) 4MS(2400) 2MS(∞)

Table 2. Automatically tuned configuration settings for the sort benchmark on various architectures. We use the following abbreviations for algorithm choices: IS = insertion sort; QS = quick sort; RS = radix sort; 16MS = 16-way merge sort; 8MS = 8-way merge sort; 4MS = 4-way merge sort; and 2MS = 2-way merge sort, with recursive merge that can be parallelized.