# “Allegro” Means Both Fast and Happy. Coincidence?

**Prepared for Italian C++ Conference 2019**

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

June 15, 2019

# Establishing Credentials

# Q: Is this guy keynote material?

**A: No.**

# Exhibit 1

## Exhibit 2

# Standard & Poor's 500 Index

# Trends in (Fast) Computing

# Pointers

- Pointer chasing is shunned
- Trees are getting f(l)atter by the year
  - Binary trees
  - Red-black trees
  - 2-3 Finger trees
  - Bitmapped vector trie—boom!
- Hashtables use open addressing instead of collision lists

## Contiguous Arrays

- Arrays++ are increasingly prevalent
- See also: *Implicit Data Structures*
- Bitmapped vector tries are more arrays than trees
- For small sizes arrays are the best data structure
  - "When you're small everything is $O(1)$"
  - Definition of 'small' grows with time

# Latency vs. Throughput

- “Low latency” relatively niche
  - Respond quickly to unpredictable events
- “Fast average throughput” the default mode
  - Ever since Z80
- Architectures optimized for tasks that are
  - Repetitive (code locality)
  - Unsurprising (speculation, prefetching)
  - Arithmetic intensive (scales well)

# Surprising Realization

Computers Love Boredom!

# I *Mean* It

- This can influence algorithm design
- Linear scan fast, random access slow
- Prefer low entropy `if`s to high entropy `if`s
  - This is highly counter-intuitive!

- Research on efficiency ahead of books on data structures and algorithms
  - Example: *Cache Oblivious Algorithms*

# What’s the Deal with Sorting?

## Sorting

- Arguably the most researched CS problem
- Many algorithms include sorting as a step
- Staple of programming classes

- Every programmer must implement sort

# Why is Quicksort Popular?

- Fundamentally easy to code and analyze
  - Can ‘spend’ on corner cases, optimization
- Fast on average
  - Just like computers!
- Little work on (almost) sorted data
  - “Idempotence should be cheap”
- Cache friendly on large inputs
- Well balanced across primitive operations

# Naïve Implementation

```cpp
template<typename It>
void qsort(It first, It last) {
  while (last - first > 1) {
    auto cut = partition_pivot(first, last);
    qsort(cut, last);
    last = cut;
  }
}
```

# Less Naïve Implementation

```
template<typename It>
void qsort(It first, It last) {
  while (last - first > THRESHOLD) {
    auto cut = partition_pivot(first, last);
    qsort(cut, last);
    last = cut;
  }
  small_sort(first, last);
}
```

- `THRESHOLD` is usually 16-32

# Challenge

- Usual `small_sort`: optimistic linear insertion sort
- Large inputs: “solved” (with caveats)
- Small inputs: “solved”
  - Optimal solutions known for $\leq 15$ elements
  - However, code size remains an issue
- Medium inputs: *difficult*

- Challenge: essentially increase `THRESHOLD`

# Investigating `small_sort`

- `std::sort`: optimistic insertion sort
- Starting from the left:
  - Find position of current in sorted subarray on the left
  - Insert it there
- Worst: $C(n) = S(n) = \frac{n(n-1)}{2}$
- Average: $C_a(n) = S_a(n) = \frac{n(n-1)}{4}$
- $C_a(32) = 248$

# Why Not *Binary* Insertion Sort?

- Same number of moves
- $C(n) = \sum_{i=1}^{n-1} \lceil \log_2 i \rceil$
- $C(32) = 155$ (compare to $248$)
- Less work for same result $\Rightarrow$ win!

## Looking Good

- Test on 1M random `double`s, threshold 32
- `std::sort`: 25.33M comparisons
- With binary insertion: 22.14M comparisons
- Cool, 15% reduction of comparisons!
- Same number of moves (13.79M)

# Oopsies

- `std::sort`: 60.75 ms
- With binary insertion: 68.58 ms
- “Cool,” 13% pessimization!
- Increasing `THRESHOLD` only makes it worse

# Some Like It Boring

- Linear searches are highly predictable
  - Literally one fail per search
  - Average success: $R(n) = \frac{n-4}{n}$, $R(32) = 87.5\%$
  - Branch prediction works swimmingly
- Binary searches have maximum entropy
  - Each extracts 1 bit of information
  - Average success: $R(n) = 50\%$
  - Branch prediction is powerless

# Unpleasant Realization

- All research: minimize $C(n)$
- All textbooks: minimize $C(n)$
- Extracting max info per comparison is a central goal

- Reality: High informational entropy of comparison affects performance

# Contrarian to the Contrarian View

- See “Binary Search Eliminates Branch Mispredictions”
- Concludes repeated binary search faster
- Specialized case (search only, powers of 2)
- Entirely/mostly inlined

# What to Do?

# "I Want Someone Smart but also Boring"

- Reducing swaps may be more productive
- Idea: start from the *middle* and expand
- Swap left with right if left $>$ right
- Insert from left
- Insert from right
- . . . until done

- Advantage: fewer swaps

# Middle-Out Insertion Sort

```
template <class It>
void middle_out_sort(It first, const It last) {
  const size_t size = last - first;
  if (size <= 1) return;
  first += size / 2 - 1;
  auto right = first + 1 + (size & 1);
  for (; right < last; ++right, --first)
  {
    if (*first > *right) swap(*first, *right);
    unguarded_linear_insert(right);
    unguarded_linear_insert_right(first);
  }
}
```

# (Aside)

- This is not terribly original
- Cottage industry of insertion sort variations
    - Two at a time insertion
    - Shell sort
    - binary merge sort
    - library sort

# Hold On To Your Hat

- Test on 1M random `double`s, threshold 32
- Middle-out insertion:
  - 23.75M comps (7% better)
  - 12.15M moves (13% better)
- However, time is identical within 1%
- Changing threshold does not help

# Hai più idee?

# Going the Other Way

- Computing systems are unfathomably complex
- Optimization is complicated and surprising
- Doing something sensible had opposite effect
- We often try clever things that don’t work
- How about trying something silly then?

# Tip: Try Silly Things

- Showerthought: worst case for insertion is moving elements over large distances
- Silly idea:

```
make_heap(begin, end, greater<>());
unguarded_insertion_sort(begin + 2, end);
```

- Like Smoothsort, just stupid
- Many comparisons still predictable
- Fewer swaps

"Oh. Well, yuk. Shellsort was that same sort of idea but both simpler and probably better."

— Mathematician upon hearing this idea

# Counts—Not Bad!

- Test on 1M random `double`s, threshold 32
- `std::sort`: 25.33M comparisons, 13.79M swaps
- Heapify+insertion sort: 23.51M comparisons, 11.56M swaps
- Improvement: 9% comps, 20% swaps

# Net Optimization

- `std::sort`: 60.54 ms
- Heapify+insertion sort: 65.92 ms
- Disaster! (9% pessimization)
- Recall we improved all metrics significantly

- Researcher's view
  - We do too many ancillary operations
  - We can increase `THRESHOLD`

# Classic heapify (Rosetta Code) 1/2

```
void to_heap(vector<int>& arr) {
  int i = (arr.size() / 2) - 1;
  for (; i >= 0; --i)
    shift_down(arr, i, arr.size());
}
```

# Classic heapify (Rosetta Code) 2/2

```
void shift_down(vector<int>& heap, int i, int max) {
  while (i < max) {
    auto i_big = i;
    auto c1 = (2 * i) + 1;
    auto c2 = c1 + 1;
    if (c1 < max && heap[c1] > heap[i_big])
      i_big = c1;
    if (c2 < max && heap[c2] > heap[i_big])
      i_big = c2;
    if (i_big == i) return;
    swap(heap[i], heap[i_big]);
    i = i_big;
  }
}
```

# Classic heapify

- Inner loop: 5 compare/jump decisions
- 3 add/shift
- 6 assignments (max)
- Must reduce this

# GNU heapify

```
template<typename _RandomAccessIterator, typename _Distance, typename _Tp, typename _Compare>
void __adjust_heap(_RandomAccessIterator __first, _Distance __holeIndex,
    _Distance __len, _Tp __value, _Compare __comp) {
  const _Distance __topIndex = __holeIndex;
  _Distance __secondChild = __holeIndex;
  while (__secondChild < (__len - 1) / 2) {
    __secondChild = 2 * (__secondChild + 1);
    if (__comp(__first + __secondChild, __first + (__secondChild - 1)))
      __secondChild--;
    *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first + __secondChild));
    __holeIndex = __secondChild;
  }
  if ((__len & 1) == 0 && __secondChild == (__len - 2) / 2) {
    __secondChild = 2 * (__secondChild + 1);
    *(__first + __holeIndex) = _GLIBCXX_MOVE(*(__first
                  + (__secondChild - 1)));
    __holeIndex = __secondChild - 1;
  }
  __decltype(__gnu_cxx::__ops::__iter_comp_val(_GLIBCXX_MOVE(__comp)))
    __cmp(_GLIBCXX_MOVE(__comp));
  std::__push_heap(__first, __holeIndex, __topIndex,
    _GLIBCXX_MOVE(__value), __cmp);
}
```

# GNU heapify

- Uses moves instead of swaps
- Special cases the sibling-less last leaf
- Inner loop: 2 compare/jump, 4 arith, 2 assign
- Outer loop: large fixup code to handle last node

# Optimization is Imagination

- Move fixup code outside outer loop
- Integrate conditionals as arithmetic
- Take advantage of the min-heap property
  - No need for bounds checks!
- Debate every penny like the lawyer of an insurance company
  - Fight for every cycle in the inner loop!

# Let’s Do This

```
template<typename It>
void insertion_sort_heap(It first, It last) {
  assert(first < last); // 0 size handled outside
  const size_t size = last - first;
  if (size < 3) {
      sort2(first[0], first[size == 2]);
      return;
  }
  heapify(first, size);
  unguarded_insertion_sort(first + 2, last);
}
```

# Heapifying

```
template<typename It>
void heapify(const It first, const size_t size) {
  assert(size > 2); // other sizes handled outside
  size_t parent = size / 2 - 1;
  do { ... inner loop ... }
  while (parent-- > 0);
  if (size & 1) return;
  for (auto i = size - 1; i > 0; ) {
    const auto parent = (i - 1) / 2;
    if (first[parent] <= first[i]) break;
    swap(first[i], first[parent]);
    i = parent;
  }
}
```

# Inner Loop

```
for (auto dad = parent;;) {
  const auto rightKid = dad * 2 + 2;
  if (rightKid >= size)
    break;
  const auto bestKid = rightKid -
    (first[rightKid] > first[rightKid - 1]);
  if (first[dad] <= first[bestKid])
    break;
  swap(first[dad], first[bestKid]);
  dad = bestKid;
}
```

# Analysis

- 3 comparisons but only 2 compare/jump
  - GNU ends up doing more work
- 2 arith, 2 assigns

- Let's put this to test!

# Meh

- Test on 1M random `double`s, threshold 32
- `std::sort`: 60.54 ms
- `insertion_sort_heap`: 61.85 ms
- Getting close (2%) but not breaking even

- But wait...
- We can increase `THRESHOLD` without compromising counts

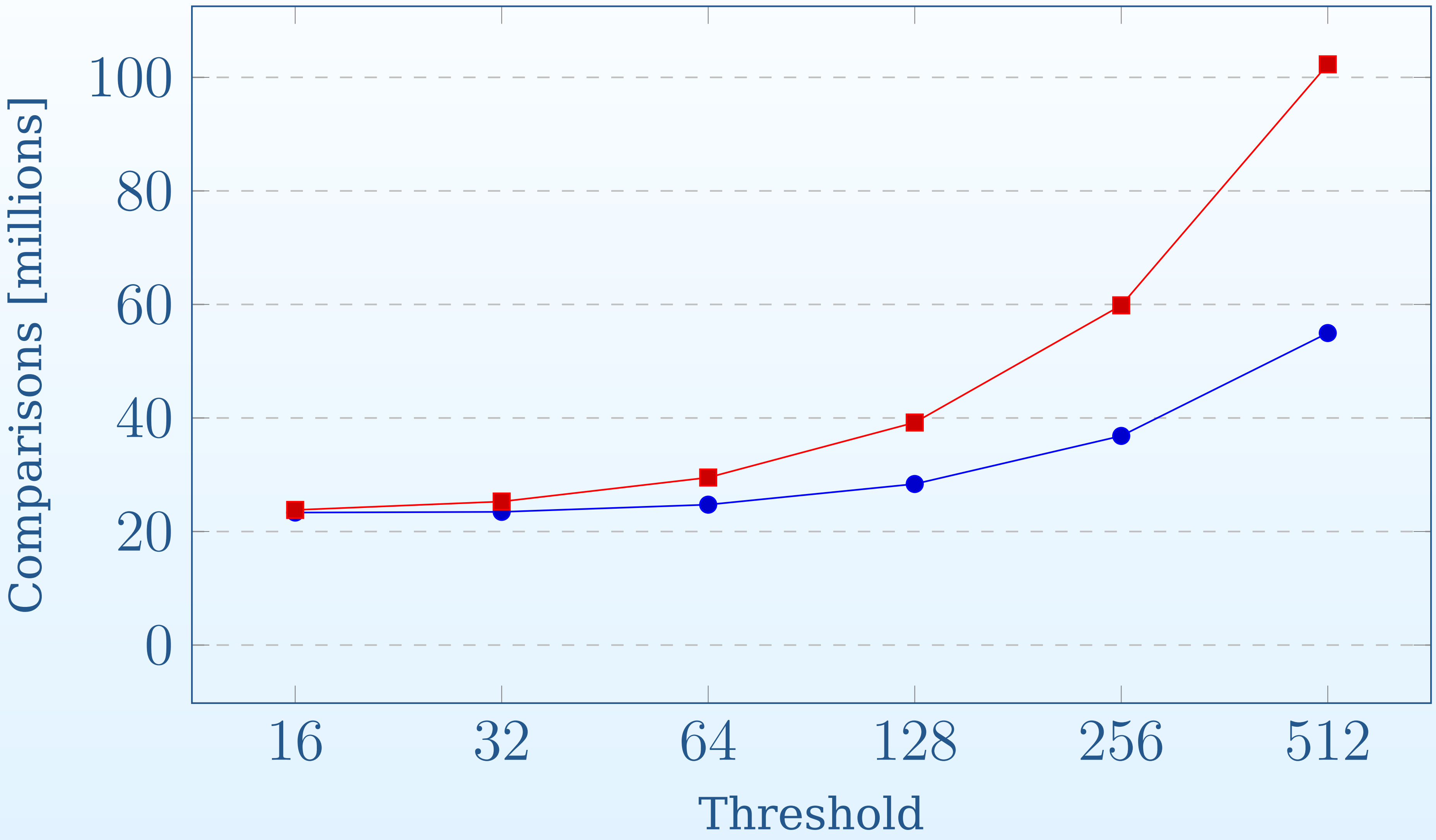
# Finally! A Significant Win

- `THRESHOLD=64`
- Reduces comparisons by 2% (for all types)
- Reduces swaps by 1.5% (for all types)
- Reduces runtime by 3% (for `double`)
  - Other types only get better

“It was exactly at that time when the talk crossed into weird territory.”

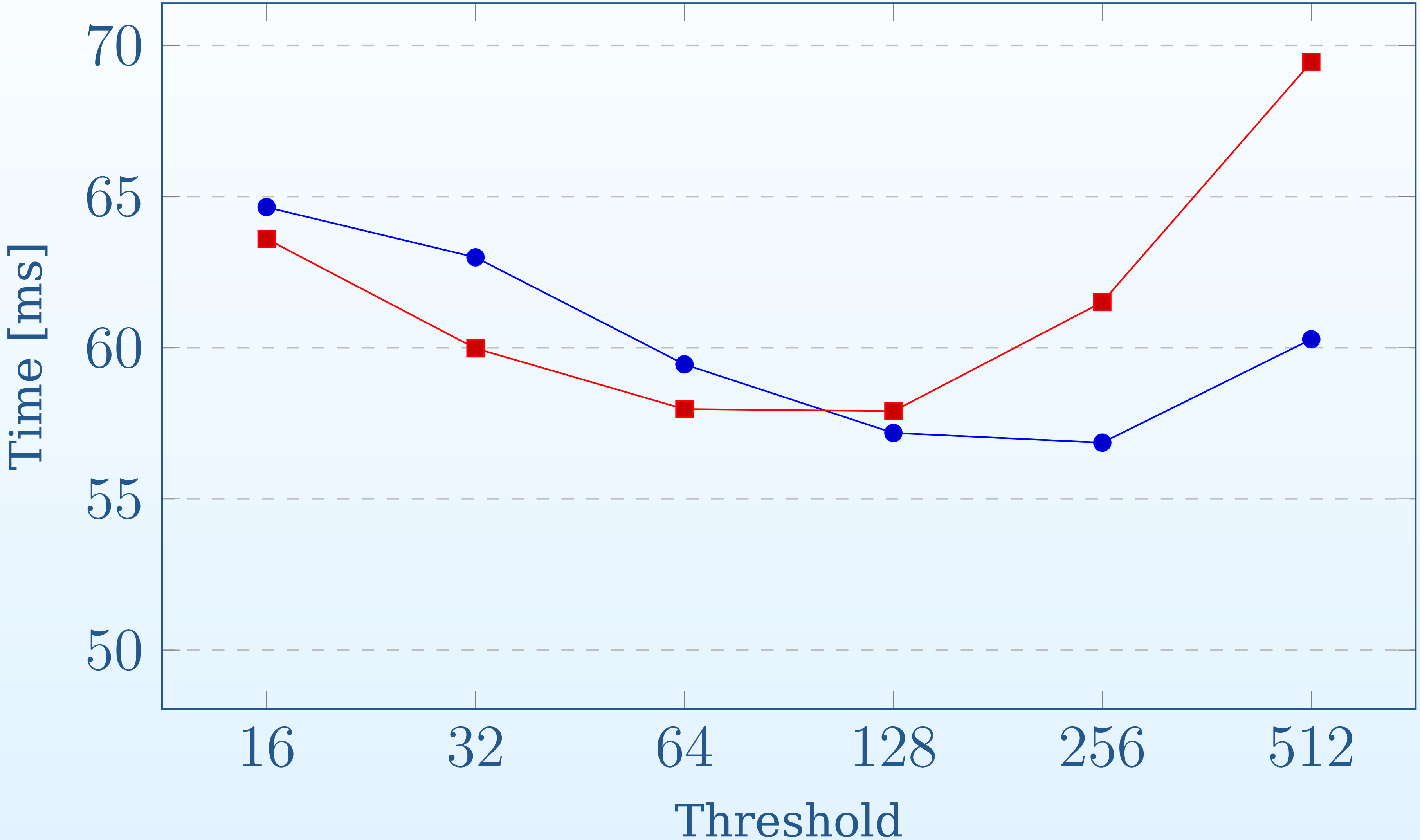
— C++ Italy Conference Attendee

# Comparisons (baseline: red)



# Moves

# Time



# Trying Silly Things

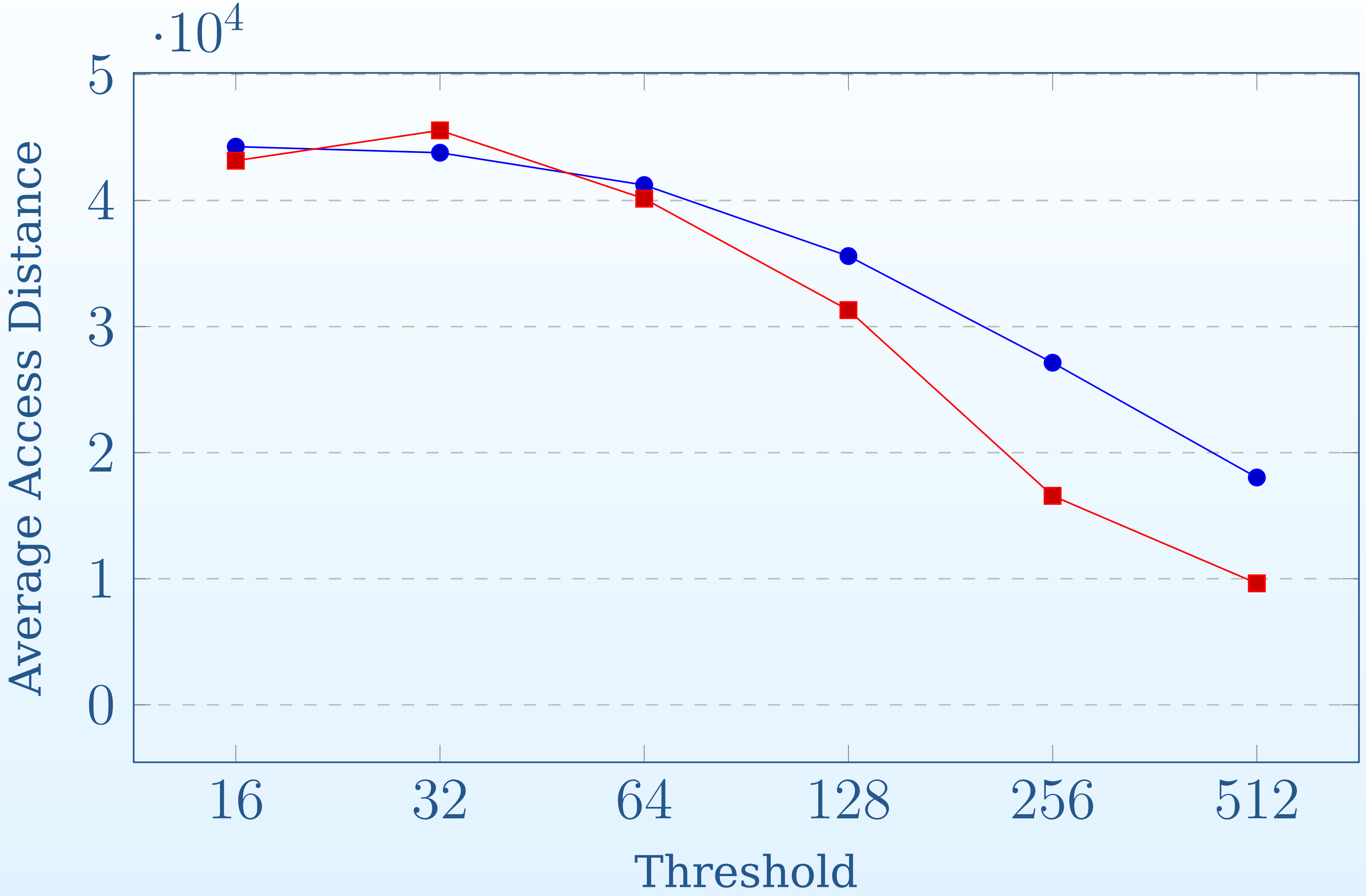
- Increasing `THRESHOLD` further:
  - Comparisons increase
  - Swaps increase
  - Time *continues to drop*
- Sweet power of 2 (for `double`): `THRESHOLD=256`
  - 36.84M comparisons (45% worse than baseline)
  - 27.21M moves (97% worse)
  - Time: **56.86 ms** (*6% better*)

# A Helpful Metric

- Collect $D(n)$, average distance between two subsequent array accesses
- A proxy for (non-)locality of array access
- Quicksort: $D(n)$ is large
- Insertion sort, heap+insertion sort: $D(n)$ is smaller
- $D(n)$ decreases as `THRESHOLD` increases

- $C(n)$ and $S(n)$ don't tell the whole story
- $D(n)$ helps independently of cache particulars

# Average Access Distance



# Summary

- Divergence between established theory and practice
- Try silly things
- Measure, measure, measure
- Devise and track meaningful metrics

# Don’t Forget

Speed is Found in the Minds of People