Modern programming styles vs. performance

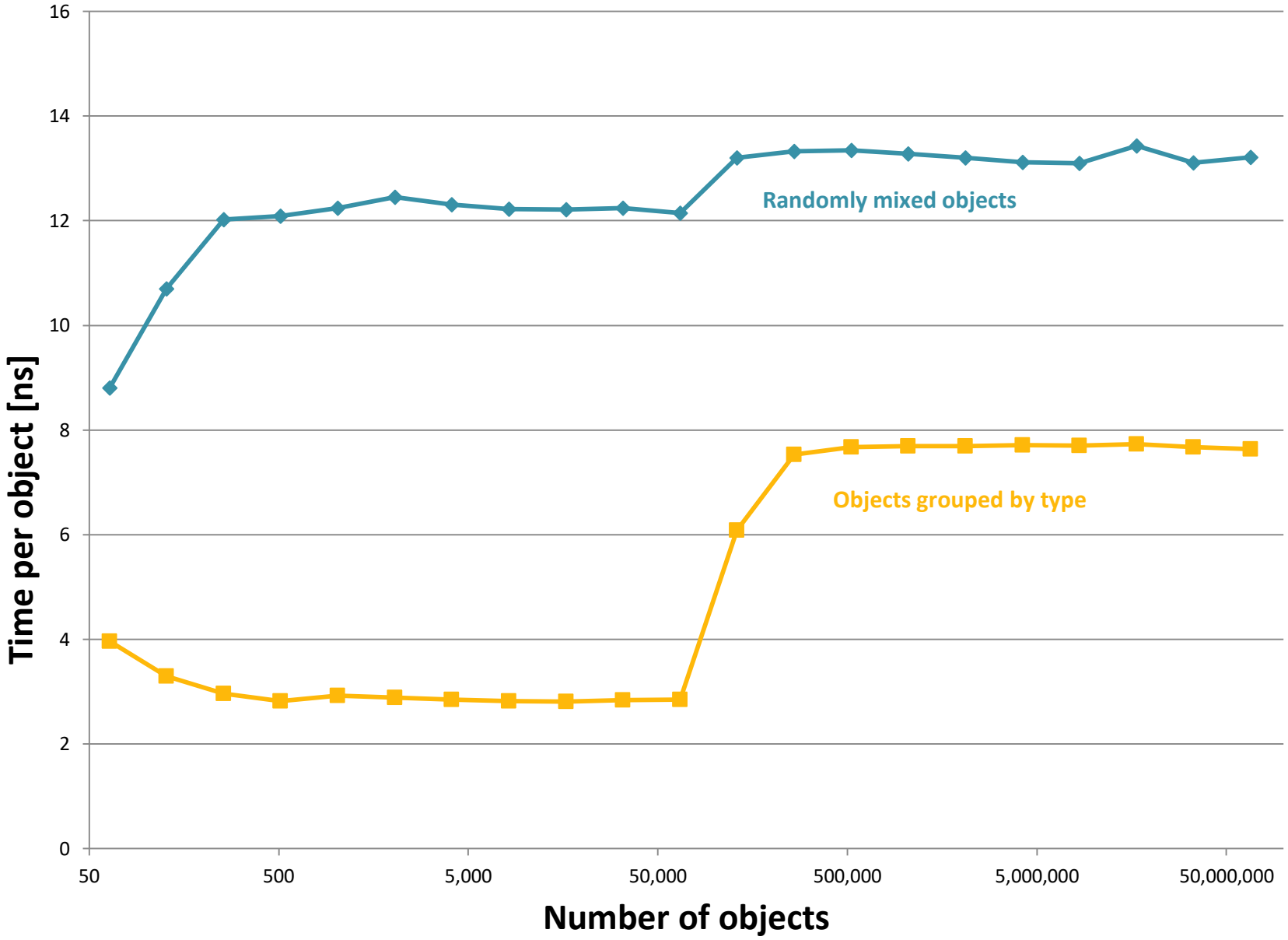
Virtual functions

Virtual functions - example

```
class A { public:
  virtual int f() = 0;
};
class C3 : public A {
  virtual int f() { return d2; }
  int d1, d2, d3;
};
class C5 : public class Fruit {
  virtual int f() { return d3; }
  int d1, d2, d3, d4, d5;
};
vector< A *> data;


int s = 0;
for_each( data.begin(), data.end(),
  [&]( A * p) { s += p->f(); });
```

- A container containing pointers to objects of several types
  - Different implementations of f()
  - Sum the values returned from f()
- Experiment setup
  - 5 concrete classes, varied size
  - f() returns a data field at different positions
  - A container filled with random mixture

Results (Intel Core Microarchitecture, 2.66 GHz)



Non-object implementation - example

```
class C3 { public:
  int f() { return d2; }
  int d1, d2, d3;
};
class C5 { public:
  int f() { return d3; }
  int d1, d2, d3, d4, d5;
};
vector< C3> data3;
vector< C5> data5;

int s = 0;
for_each( data3.begin(), data3.end(),
  [&]( C3 & p) { s += p.f(); });
for_each( data5.begin(), data5.end(),
  [&]( C5 & p) { s += p.f(); });
```

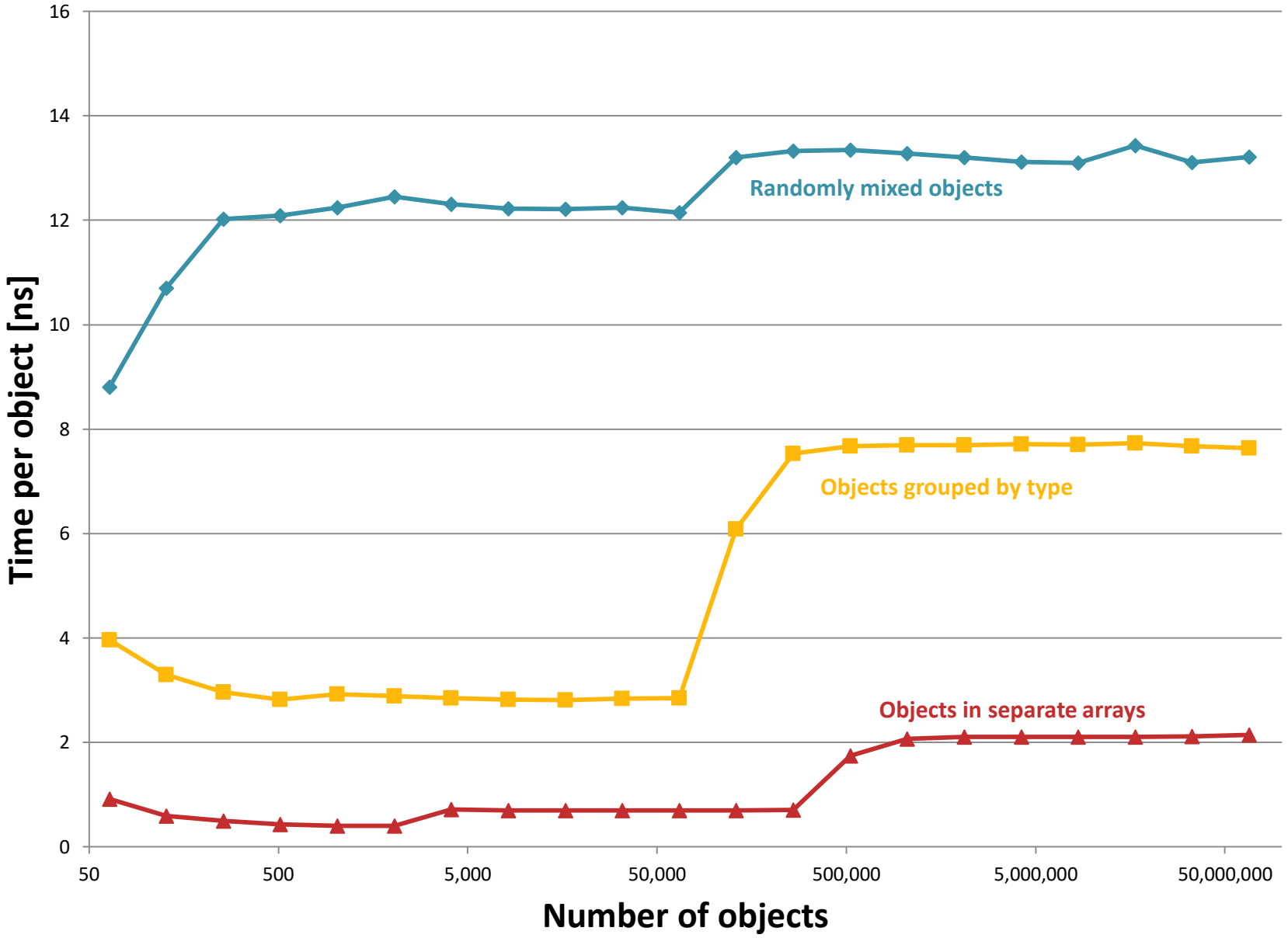
- No base class, no virtual functions
- No pointers required

- Several containers required
  - Difficult to extend with new object types

```
std::tuple<std::vector<TLIST>...>
```

- Does not preserve the order of objects
  - If required, an additional array of indexes is needed
  - For non-commutative aggregation, any advantage is lost

Results (Intel Core Microarchitecture, 2.66 GHz)

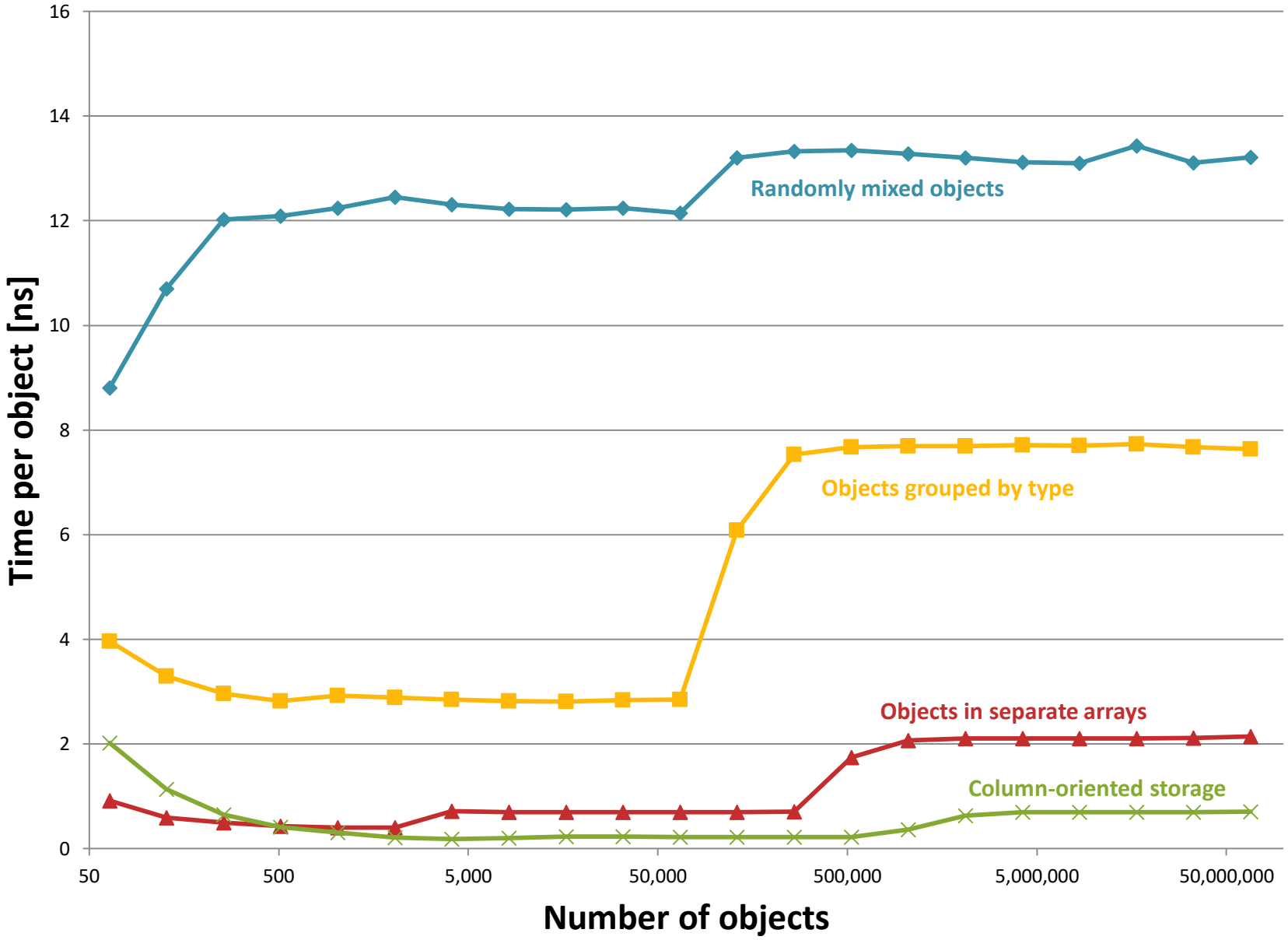


Column-oriented implementation - example

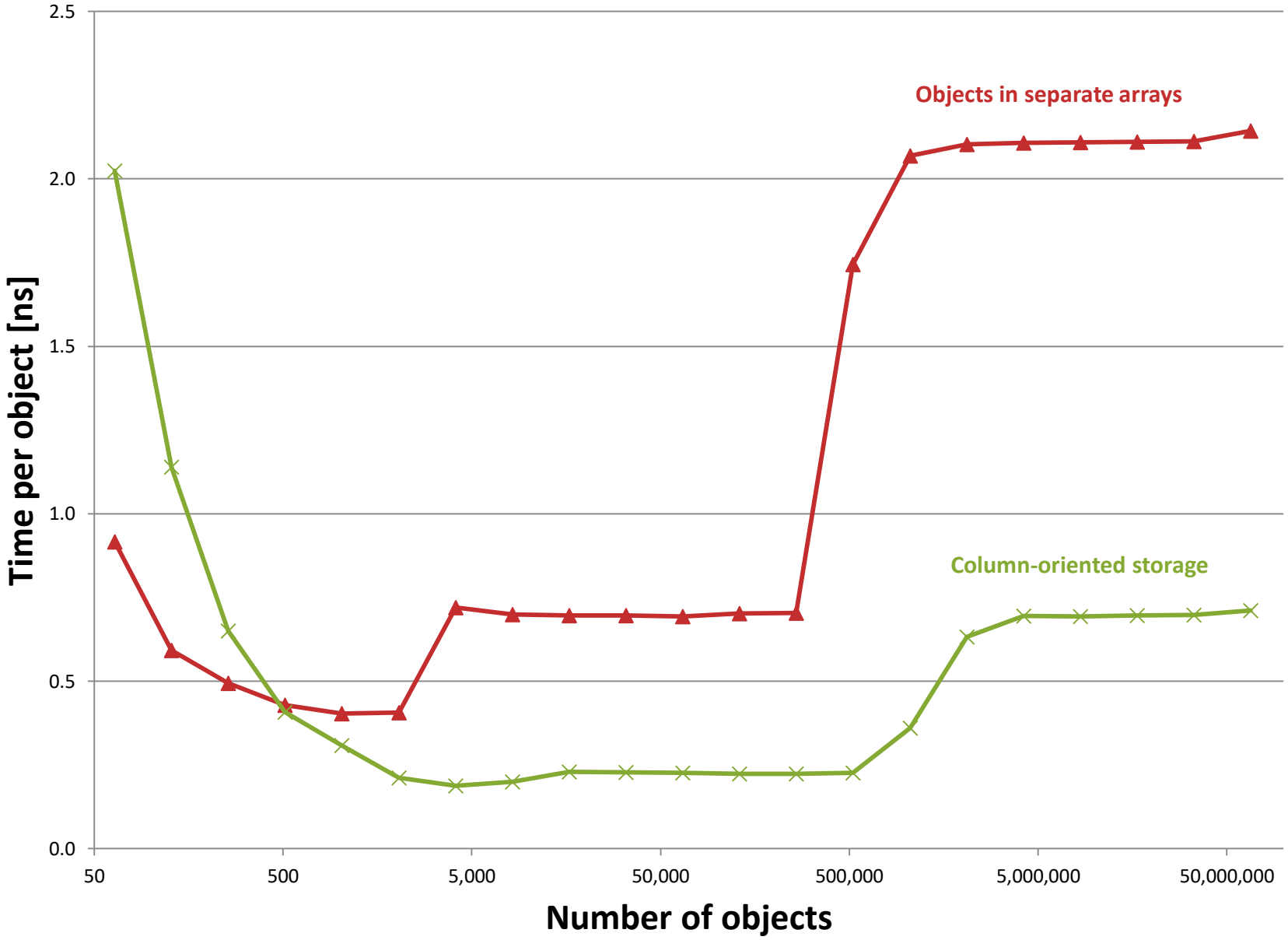
```
vector< int> data3d1;
vector< int> data3d2;
vector< int> data3d3;
vector< int> data5d1;
vector< int> data5d2;

vector< int> data5d3;
vector< int> data5d4;
vector< int> data5d5;

int s = 0;
s += std::reduce( data3d2.begin(),
  data3d2.end(), 0, std::plus<int>());
s += std::reduce( data5d3.begin(),
  data5d3.end(), 0, std::plus<int>());
```

- No classes at all
- Just arrays of elementary types
  - Extremely difficult and error-prone
  - No support for generic programming in this style
- For each column, data are dense
  - Improves cache behavior for column-oriented access
  - Degrades cache behavior for row-oriented access
  - Allows vectorization
    - std::reduce [C++17] allows compilers to vectorize automatically
    - The experiments were done with manual vectorization

Results (Intel Core Microarchitecture, 2.66 GHz)



Results (Intel Core Microarchitecture, 2.66 GHz)



Summary

- Grouping the data by type
  - No change in code
  - More than **4**-fold speed-up (with 10000 objects)
- C-style programming
  - Separate arrays of structures
  - Additional **4**-fold speed-up
- FORTRAN-style programming
  - Arrays of elementary types
  - Vectorization enabled
  - Additional **3**-fold speed-up
- FORTRAN vs. Object-oriented programming: **50 : 1**
  - Fast but breaking all software-engineering rules
- Warning: This is a well-designed example
  - Very simple operations, order not preserved, …