Colliding Blobs
with Threading Building Blocks

Adam Sampson
Institute of Arts, Media and Computer Games
University of Abertay Dundee

Abertay University

Motivation

- MSc projects this summer simulating physical interactions between cells in a tissue
  - All-pairs, computing forces between elements
  - … at least to start with
- They're interested in parallelising it, but they've not done any parallel programming before... how well is this likely to work?
- Try a *really simple* approach to parallelisation – what the tutorials tell you to do!

Implementation

- All-pairs nbody in C++0x
- Write readable code and see how well the compiler does
  - … but I'll measure this later
  - Hints: inlining, const annotations...
- Liberal use of the standard library and of Boost
- 3D vector class
- All templated over scalar/vector types:
  `universe<vec3<float>>`

Benchmarking

- Benchmarked on several different machines
- run-tests script for automated benchmarking
  - Vary compiler options
  - Vary runtime options
  - Vary number of threads
  - Produce data and config files for gnuplot
- Ensured no memory pressure, and profiled to confirm I was timing the appropriate bit
  - … not very hard with this problem!

Compiler options

- Tune for appropriate architecture
  - `-march=core2`, etc. (implies `-mtune`)
- Try 387 maths vs. SSE maths
  - `-mfpmath=387`, `-mfpmath=sse`
- Try `-O2`, `-O3`, `-Os`
  - Optimising for size used to be a good idea on cache-starved CPUs...

Vector representation

- Conventional implementation, templated over scalar type (both float and double)

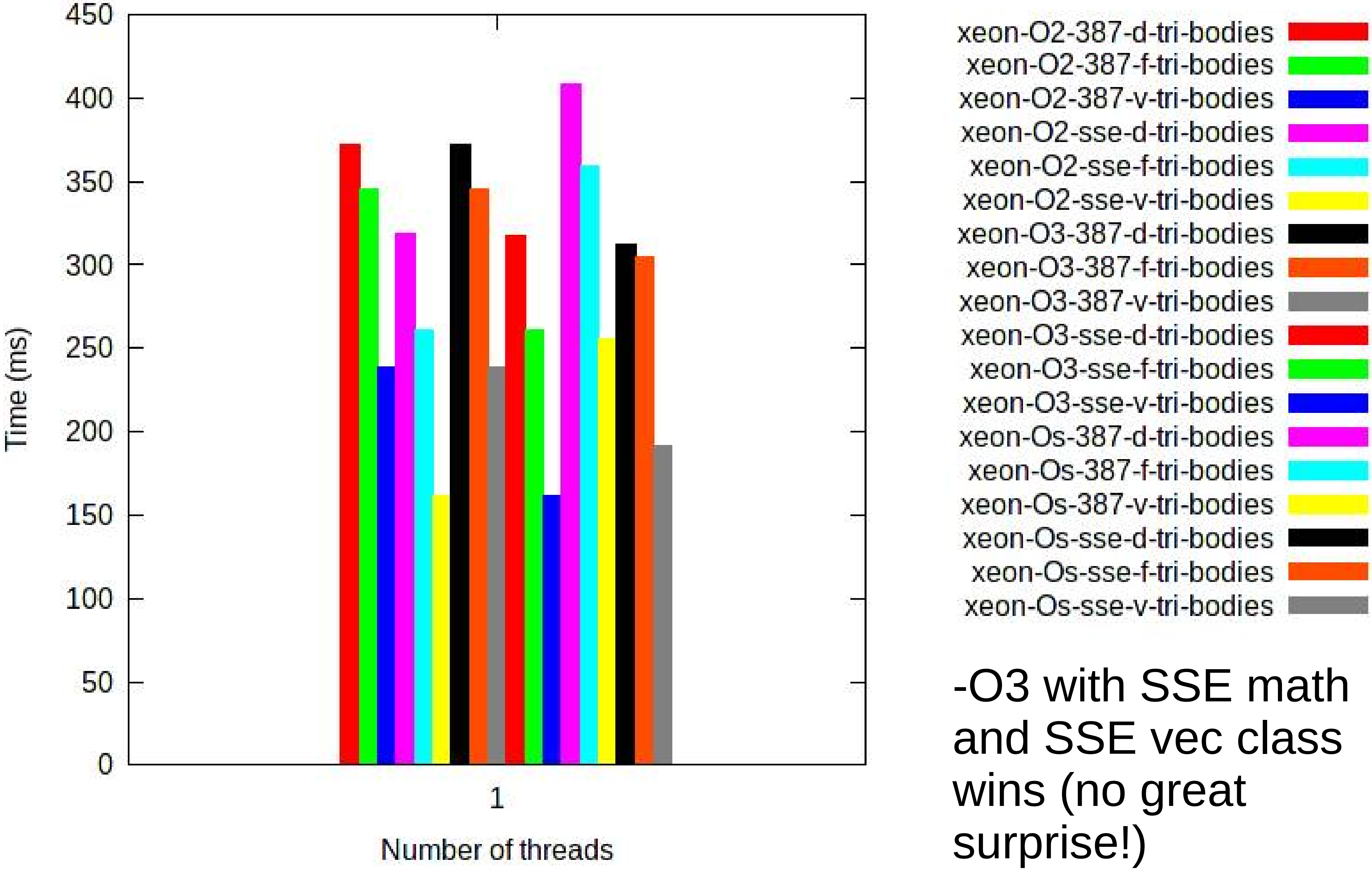
```
template<typename T>
class vec3 {
        ...
        vec3<T>& operator+=(const vec3<T>& o) {
                x_ += o.x_;
                y_ += o.y_;
                z_ += o.z_;
                return *this;
        }
        ...
```

Vector representation

- … or implementation using the SSE intrinsics
- Alignment problems with `std::vector`
  - Use `tbb::cache_aligned_allocator`

```
class vec {   // just a _m128 really
        ...
        vec& operator+=(const vec& o) {
                v_ = _mm_add_ps(v_, o.v_);
                return *this;
        }
        ...
```

Results



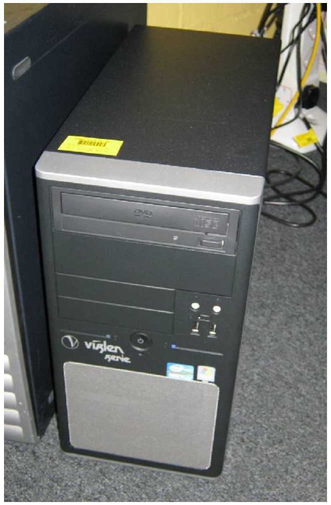
-O3 with SSE math and SSE vec class wins (no great surprise!)

An aside on std::vector

- There's a persistent myth (especially in the games world) that “the STL is slow”
  - (Note that some myths are true...)
- For a *good* compiler, this is not the case
  - `vector` should behave identically to an array...
  - VC++ is *not* a good compiler
- In the sequential nbody, GCC's optimiser inlines everything – you get one large function in the generated code

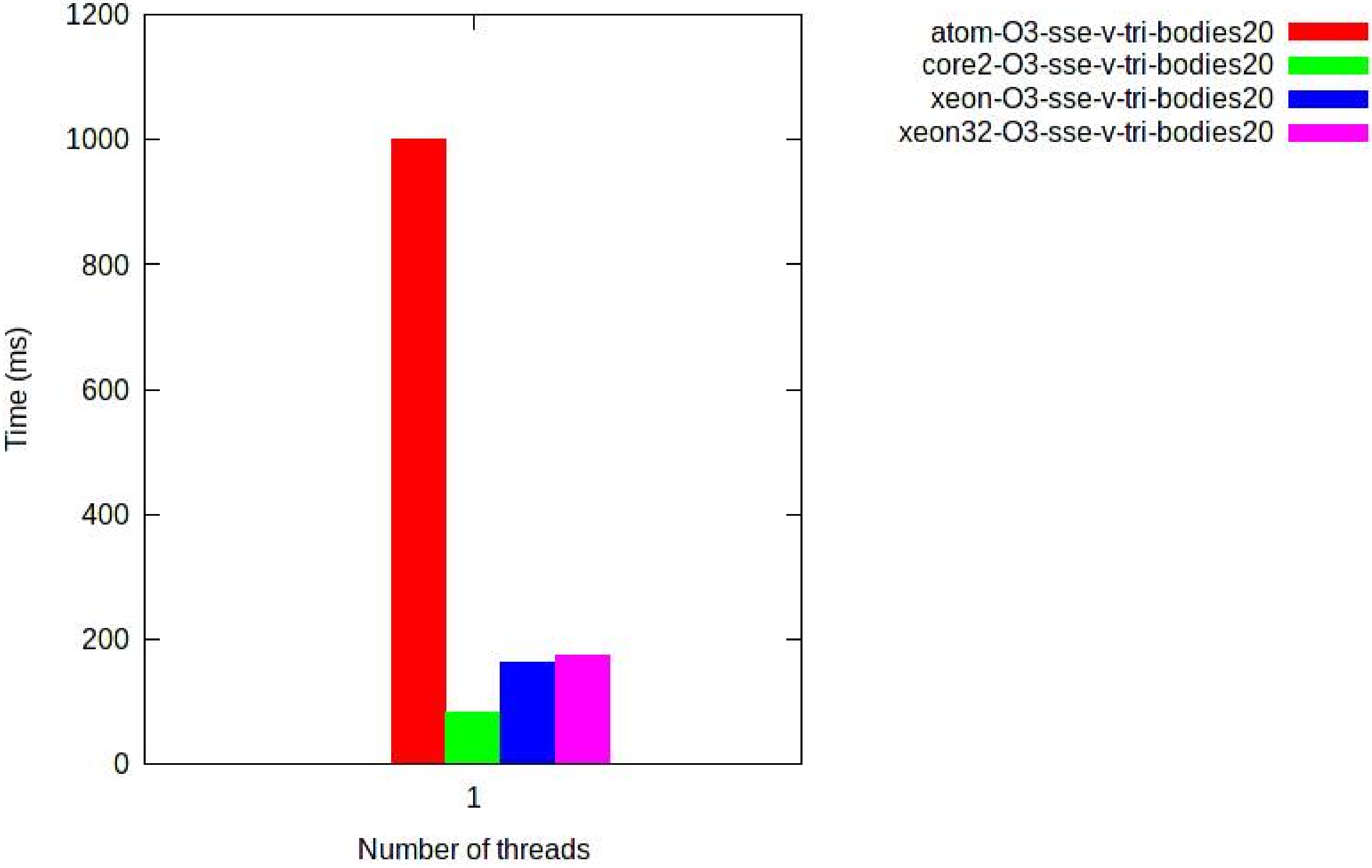
Machines

- Atom N270
  1.6GHz, 1 core
- Core i7-2600
  3.4Ghz, 4 cores
- 2x Xeon E5520
  2.27GHz. 4 cores
- All cores 2x HT
- Debian, GCC 4.4,
  TBB 3.0



Abertay University

Machine performance



Data

```
int nbodies_;
// Keep positions packed together for better cache
 // usage above.
// CAA gets us enough alignment for SSE to work.
std::vector<V, tbb::cache_aligned_allocator<V>> pos_;
std::vector<V, tbb::cache_aligned_allocator<V>> vel_;
// This doesn't need to be aligned, but it doesn't hurt.
std::vector<S, tbb::cache_aligned_allocator<S>> mass_;

// FIXME: try different storage layouts
```

Triangular advance

```
void advance_tri() {
   for (int i = 0; i < nbodies_; ++i) {
      for (int j = i + 1; j < nbodies_; ++j) {
         V d(pos_[i] - pos_[j]);
         S distance(d.mag(soften_));
         S mag(dt_ / (distance * distance * distance));
         vel_[i] -= d * (mass_[j] * mag);
         vel_[j] += d * (mass_[i] * mag);
      }
   }

   for (int i = 0; i < nbodies_; ++i) {
      pos_[i] += vel_[i] * dt_;
   }
}
```

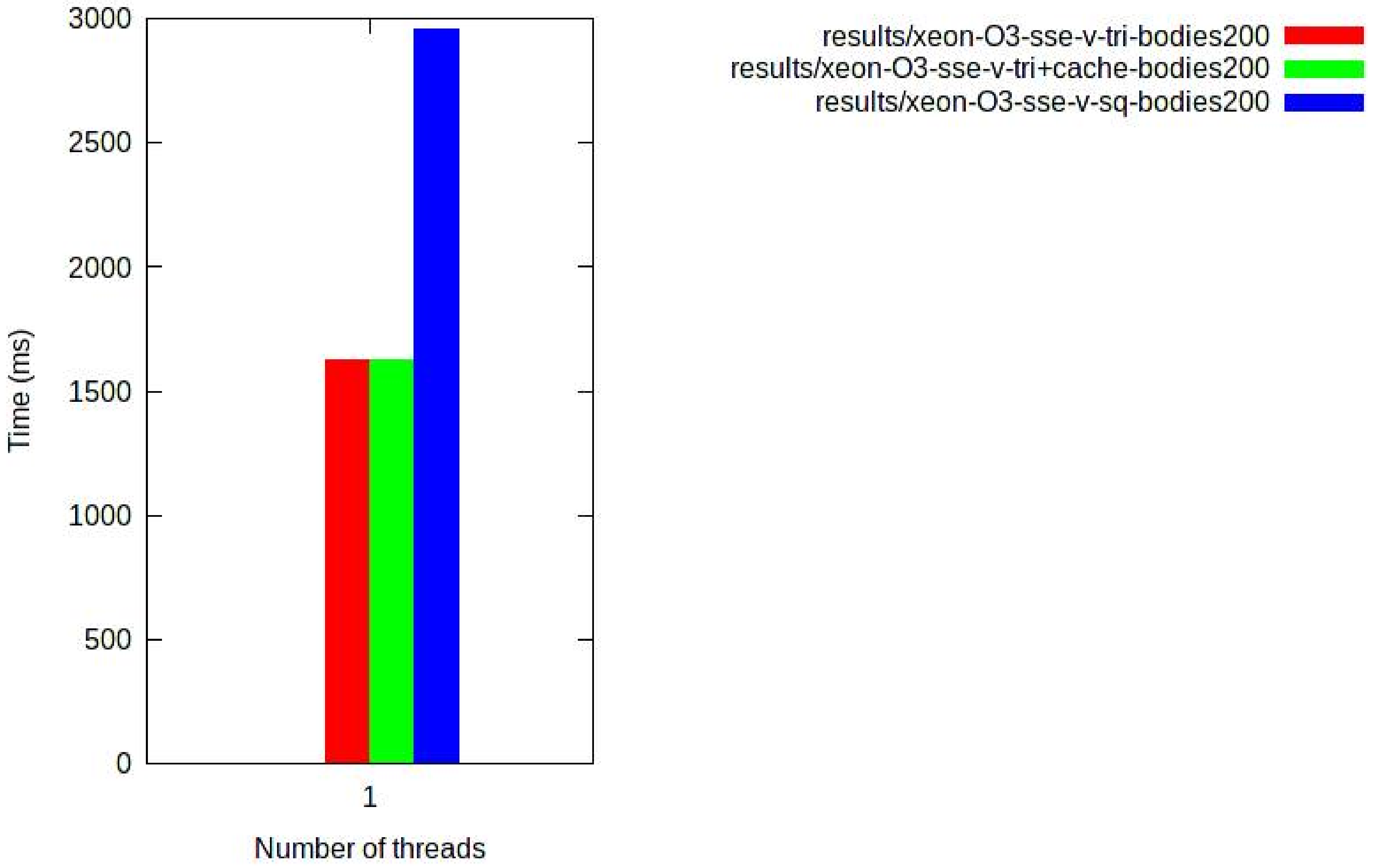
Tweaked triangular advance

```
void advance_tri_cache() {
   const S soften(soften_);
   const S dt(dt_);

   for (int i = 0; i < nbodies_; ++i) {
      for (int j = i + 1; j < nbodies_; ++j) {
         const V d(pos_[i] - pos_[j]);
         const S distance(d.mag(soften));
         const S mag(dt / (distance*distance*distance));
         vel_[i] -= d * (mass_[j] * mag);
         vel_[j] += d * (mass_[i] * mag);
      }
   }

   for (int i = 0; i < nbodies_; ++i) {
      pos_[i] += vel_[i] * dt;
   }
}
```

Square advance

```
void advance_sq() {
   for (int i = 0; i < nbodies_; ++i) {
      V vel(vel_[i]);
      for (int j = 0; j < nbodies_; ++j) {
         if (i == j) {
            continue;
         }
         V d(pos_[i] - pos_[j]);
         S distance(d.mag(soften_));
         S mag(dt_ / (distance * distance * distance));
         vel -= d * (mass_[j] * mag);
      }
      vel_[i] = vel;
   }
   for (int i = 0; i < nbodies_; ++i) {
      pos_[i] += vel_[i] * dt_;
   }
}
```

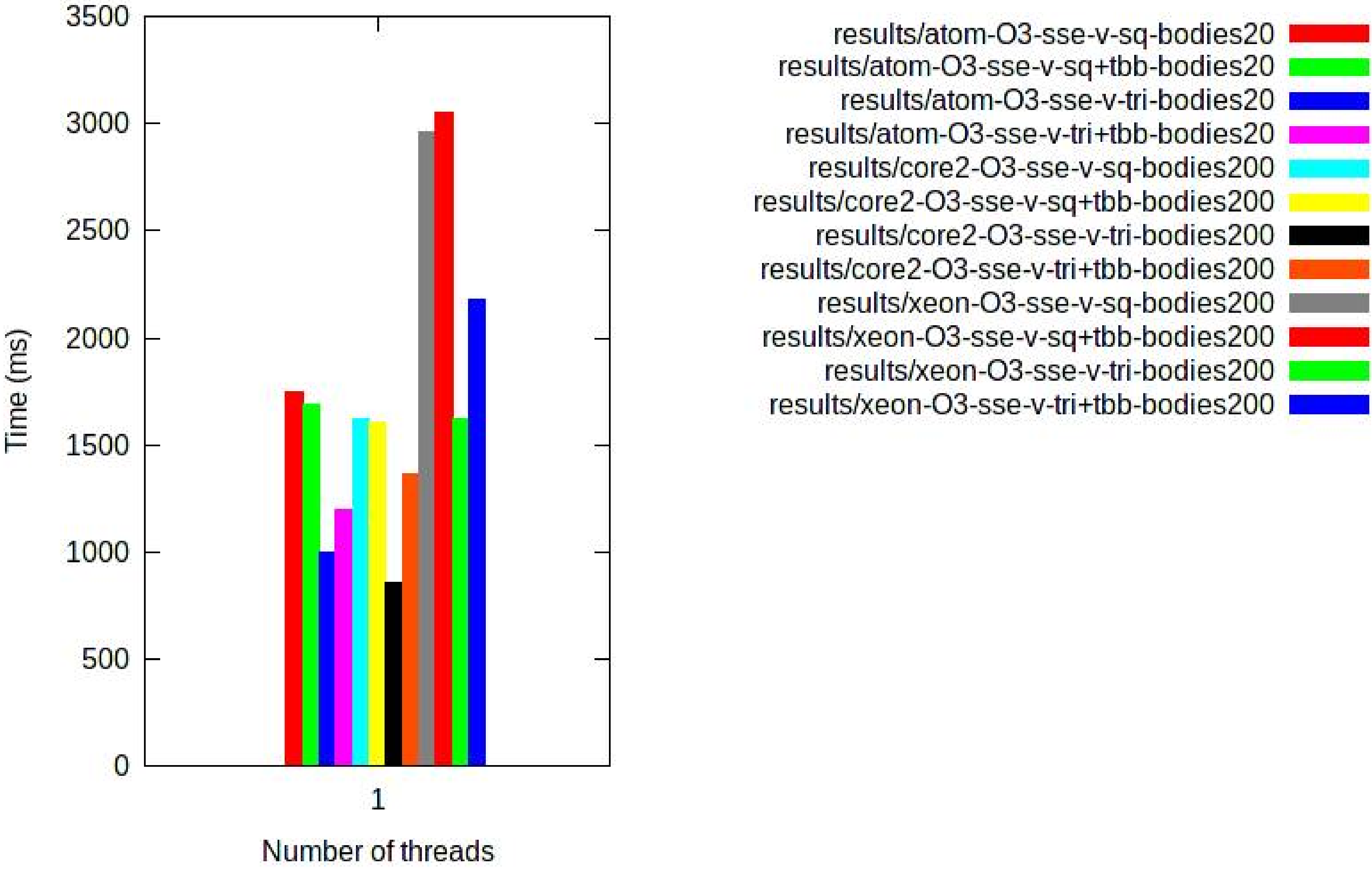
Mode results



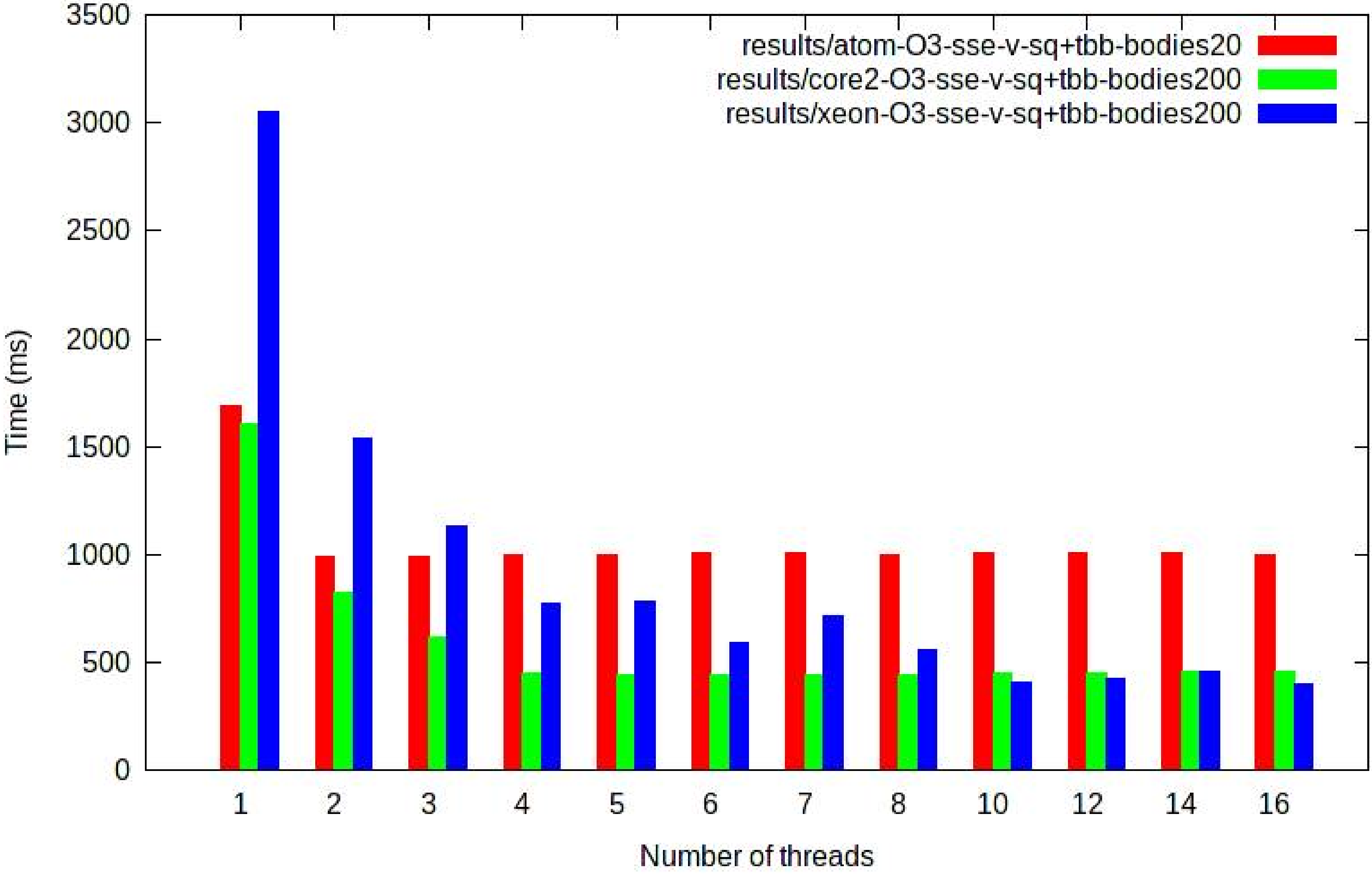
TBB square advance

```
class sq_tbb_worker {
public:
   sq_tbb_worker(universe& u) : u_(u) {}
   void operator()(tbb::blocked_range<int> &r) const {
      for (int i = r.begin(); i < r.end(); ++i) {
         ...  update velocities as before
      }
   }
private:
   universe& u_;
};
friend class sq_tbb_worker;

void advance_sq_tbb() {
   tbb::blocked_range<int> r(0, nbodies_);
   sq_tbb_worker worker(*this);
   tbb::parallel_for(r, worker);
   ...  update positions as before
```

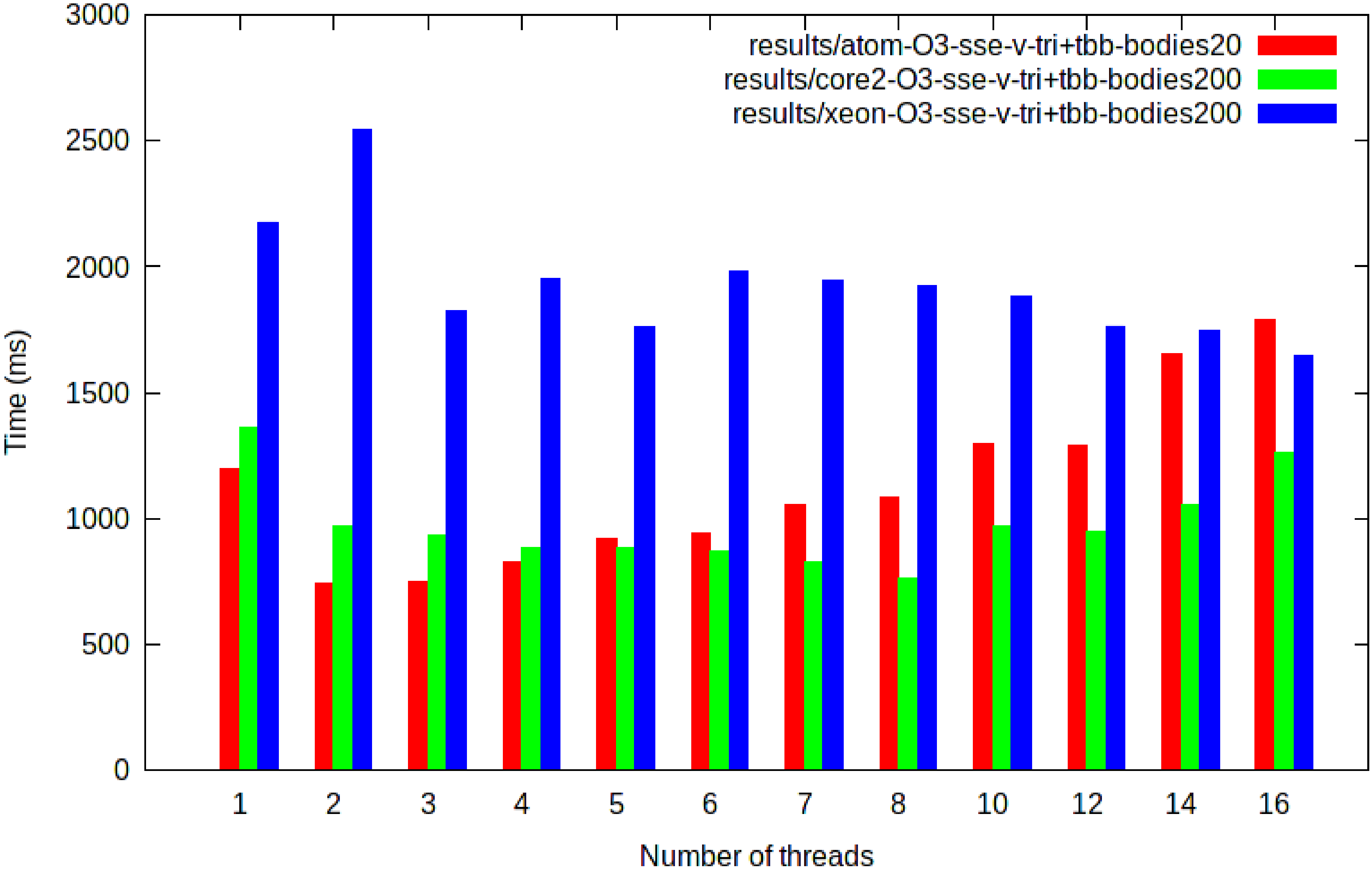
TBB vs. sequential



TBB square results



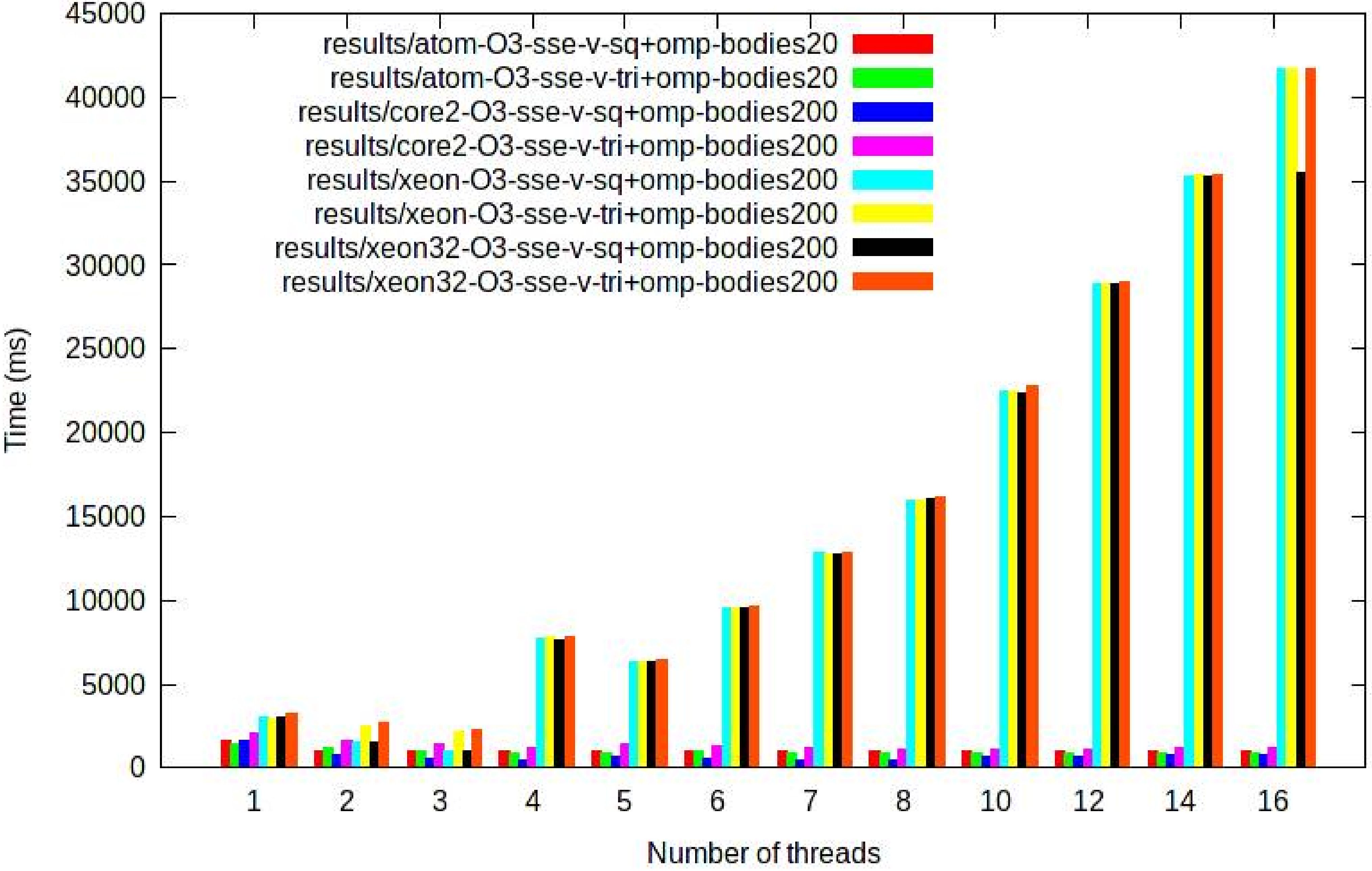
TBB triangular results – spinning



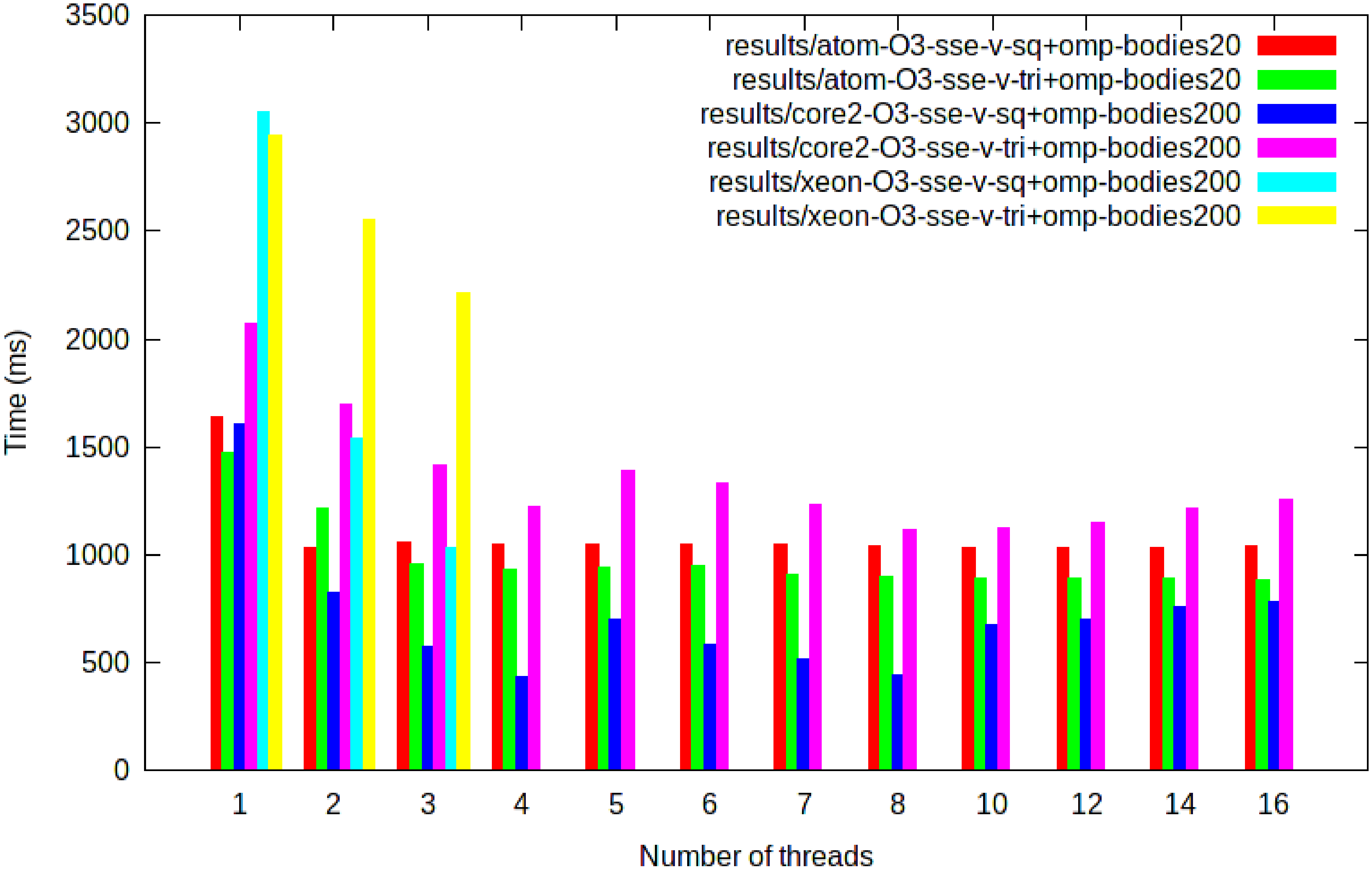
OpenMP square advance

```
  void advance_sq_omp() {
#pragma omp parallel for
    for (int i = 0; i < nbodies_; ++i) {
      V vel(vel_[i]);
      for (int j = 0; j < nbodies_; ++j) {
        if (i == j) {
          continue;
        }
        V d(pos_[i] - pos_[j]);
        S distance(d.mag(soften_));
        S mag(dt_ / (distance * distance * distance));
        vel -= d * (mass_[j] * mag);
      }
      vel_[i] = vel;
    }
    for (int i = 0; i < nbodies_; ++i) {
      pos_[i] += vel_[i] * dt_;
    }
  }
```

OpenMP results – argh!



OpenMP results trimmed



Any questions?

- Thanks for listening!
- Get the code:
  git clone http://offog.org/git/sicsa-mcc.git
- Contact me or get this presentation:
  http://offog.org/
- Threading Building Blocks
  http://threadingbuildingblocks.org/