OpenMP case studies

Victor Eijkhout

Case study: molecular dynamics

Formulation

A particle has x, y coordinates and a mass c. For two particles (x_1, y_1, c_1), (x_2, y_2, c_2) the force on particle 1 from particle 2 is:

$$\overrightarrow{F}_{12} = \frac{c_1 \cdot c_2}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \cdot \overrightarrow{r}_{12}$$

where $\overrightarrow{r}_{12}$ is the unit vector pointing from particle 2 to 1. With n particles, each particle i feels a force

$$\overrightarrow{F}_i = \sum_{j \neq i} \overrightarrow{F}_{ij}.$$

Building blocks

Let's start with a couple of building blocks.

```
// molecularstruct.c
struct point{ double x,y; double c; };
struct force{ double x,y; double f; };

/* Force on p1 from p2 */
struct force force_calc( struct point p1,struct point p2 ) {
  double dx = p2.x - p1.x, dy = p2.y - p1.y;
  double  f = p1.c * p2.c / sqrt( dx*dx + dy*dy );
  struct force exert = {dx,dy,f};
  return exert;
}
```

Sequential code in C

(Probably wrong, but hey, I'm not a physicist)

```
void add_force( struct force *f,struct force g ) {
  f->x += g.x; f->y += g.y; f->f += g.f;
}
void sub_force( struct force *f,struct force g ) {
  f->x -= g.x; f->y -= g.y; f->f += g.f;
}
```

For reference, this is the sequential code:

```
for (int ip=0; ip<N; ip++) {
  for (int jp=ip+1; jp<N; jp++) {
    struct force f = force_calc(points[ip],points[jp]);
    add_force( forces+ip,f );
    sub_force( forces+jp,f );
  }
}
```

Here $\overrightarrow{F}_{ij}$ is only computed for $j > i$, and then added to both $\overrightarrow{F}_i$ and $\overrightarrow{F}_j$.

In C++

In C++ we can have a class with an addition operator and such:

```
// molecular.cxx
class force {
private:
  double _x{0.},_y{0.}; double _f{0.};
public:
  force() {};
  force(double x,double y,double f)
    : _x(x),_y(y),_f(f) {};
```

```
force operator+( const force& g ) {
  return { _x+g._x, _y+g._y, _f+g._f };
}
```

Sequential code:

```
for (int ip=0; ip<N; ip++) {
  for (int jp=ip+1; jp<N; jp++) {
    force f = points[ip].force_calc(points[jp]);
    forces[ip] += f;
    forces[jp] -= f;
  }
```

Exercise

Is the outer loop parallelizable? The inner? Both together?

Solution 1: full interactions

One solution would be to compute the $\overrightarrow{F}_{ij}$ interactions for all i, j, so that there are no conflicting writes.

```
for (int ip=0; ip<N; ip++) {
        struct force sumforce;
        sumforce.x=0.; sumforce.y=0.; sumforce.f=0.;
#pragma omp parallel for reduction(+:sumforce)
  for (int jp=0; jp<N; jp++) {
    if (ip==jp) continue;
    struct force f = force_calc(points[ip],points[jp]);
    sumforce.x += f.x; sumforce.y += f.y; sumforce.f += f.f;
  } // end parallel jp loop
  add_force( forces+ip, sumforce );
} // end ip loop
```

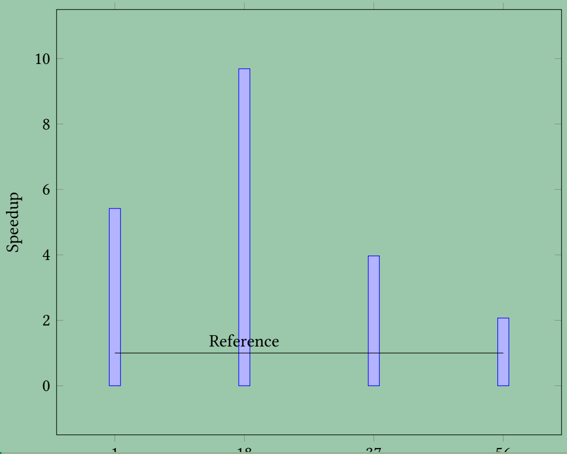
C++ variant: overloaded reduction

In C++ we use the fact that we can reduce on any class that has an addition operator:

```
for (int ip=0; ip<N; ip++) {
  force sumforce;
  #pragma omp parallel for reduction(+:sumforce)
  for (int jp=0; jp<N; jp++) {
    if (ip==jp) continue;
    force f = points[ip].force_calc(points[jp]);
    sumforce += f;
  } // end parallel jp loop
  forces[ip] += sumforce;
} // end ip loop
```

Exercise

This increases the scalar work by a factor of two, but surprisingly, on a single thread the run time improves: we measure a speedup of `6.51` over the supposedly 'optimal' code. (Why?)



Solution 2: atomic updates

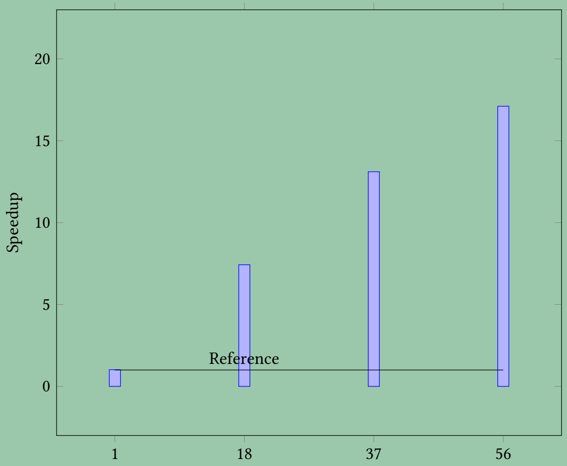
The `i` update is fine, we make the `j` update atomic:

```
#pragma omp parallel for schedule(guided,4)
      for (int ip=0; ip<N; ip++) {
        for (int jp=ip+1; jp<N; jp++) {
          struct force f = force_calc(points[ip],points[jp]);
          add_force( forces+ip,f );
          sub_force( forces+jp,f );
        }
      }
```

To deal with the conflicting *jp* writes, we make the writes atomic:

```
void sub_force( struct force *f,struct force g ) {
#pragma omp atomic
  f->x -= g.x;
#pragma omp atomic
  f->y -= g.y;
#pragma omp atomic
  f->f += g.f;
}
```

Result



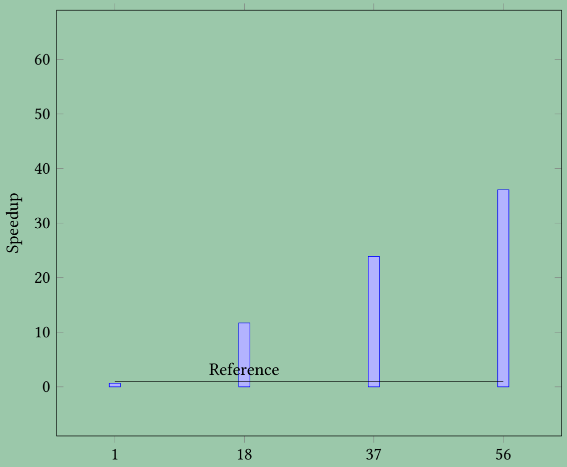
What happens with one thread?

Solution 3: fully atomic

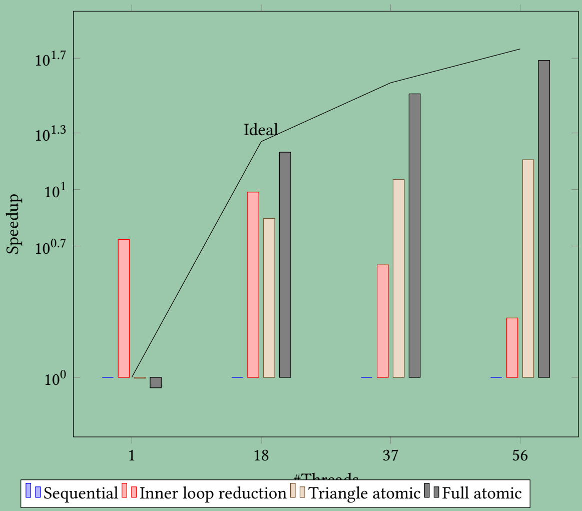
But if we decide to use atomic updates, we can take the full square loop, collapse the two loops, and make every write atomic.

```
#pragma omp parallel for collapse(2)
      for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<N; jp++) {
          if (ip==jp) continue;
          struct force f = force_calc(points[ip],points[jp]);
          add_force( forces+ip, f );
        } // end parallel jp loop
      } // end ip loop
```

Results



All results together

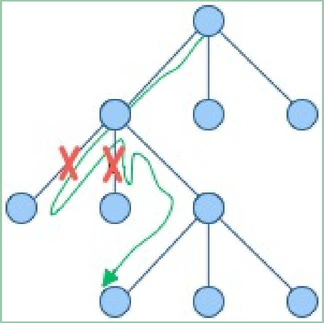


Search: 8 queens

Tree traversal

Search: traverse the tree,
and abort unsuccessful branches

DFS, not BFS



Eight queens

```
placement initial; initial.fill(empty);
auto solution = place_queen(0,initial);

optional<placement> place_queen
        (int iqueen,const placement& current) {
  for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
    if (feasible(next)) {
      if (iqueen==N-1)
        return next;
      auto attempt = place_queen(iqueen+1,next);
      if (attempt.has_value())
        return attempt;
    } // end if(feasible)
  }
  return {};
};
```

With OpenMP

```
placement initial; initial.fill(empty);
optional<placement> eightqueens;
#pragma omp parallel
#pragma omp single
eightqueens = place_queen(0,initial);
```

More tasks

We create a task for each column, and since they are in a loop we use `taskgroup` rather than `taskwait`.

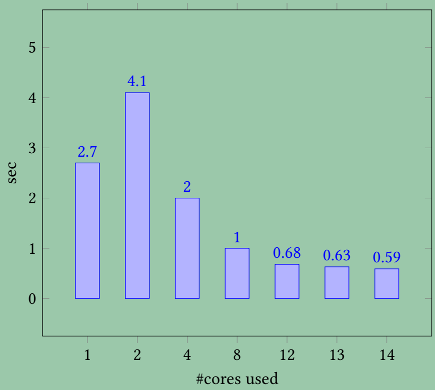
```
#pragma omp taskgroup
  for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
#pragma omp task firstprivate(next)
    if (feasible(next)) {
    // stuff
    } // end if(feasible)
  }
```

How to break

However, the sequential program had *return* and *break* statements in the loop, which is not allowed in workshare constructs such as taskgroup. Therefore we introduce a return variable, declared as shared:

```
// queens0.cxx
optional<placement> result = {};
#pragma omp taskgroup
for (int col=0; col<N; col++) {
  placement next = current;
  next.at(iqueen) = col;
  #pragma omp task firstprivate(next) shared(result)
  if (feasible(next)) {
    if (iqueen==N-1) {
        result = next;
    } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
          result = attempt;
        }
    } // end if(iqueen==N-1)
  } // end if(feasible)
}
```

Timing



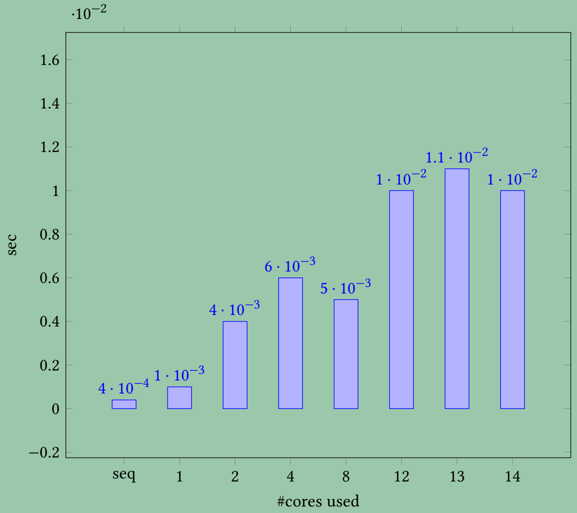
This is a 1000 times slower than sequential. Why?

Cancelling

Body of the loop over columns:

```
// queenfinal.cxx
if (feasible(next)) {
  if (iqueen==N-1) {
        result = next;
    #pragma omp cancel taskgroup
  } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
          result = attempt;
      #pragma omp cancel taskgroup
        }
  } // end if (iqueen==N-1)
} // end if (feasible)
```

Timing



Still not great. Conclusion?

C++ notes

Range syntax

Parallel loops in C++ can use range-based syntax:

```
// speedup.cxx
#pragma omp parallel for
for ( auto& v : values ) {
  for (int jp=0; jp<M; jp++) {
    double f = sin( v );
    v = f;
  }
}
```

Tests not reported here show exactly the same speedup as the C code.

Iterators

Support for *C++ iterators*

```
#pragma omp declare reduction (merge : std::vector<int>
    : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

Templated reductions

You can reduce with a templated function if you put both the declaration and the reduction in the same templated function:

```
template<typename T>
T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction                              \
  (rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))  \
  initializer(omp_priv=-1.f)

  T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
  for (int id=0; id<tdata.size(); id++)
    tmin = reduce_without_zero<T>(tmin,tdata[id]);
  return tmin;
};
```

which is then called with specific data:

```
auto tmin = generic_reduction<float>(fdata);
```

Reducing on overloaded operator

Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```
// reductcomplex.cxx
class Thing {
private:
  float x;
public:
  Thing() : Thing( 0.f ) {};
  Thing( float x ) : x(x) {};
  Thing operator+( const Thing&
      ↪other ) {
    return Thing( x + other.x );
  };
};
```

```
vector< Thing >
      ↪things(500,Thing(1.f) );
Thing result(0.f);
#pragma omp parallel for
      ↪reduction( +:result )
for ( const auto& t : things )
  result = result + t;
```

A default constructor is required for the internally used init value; see figure **??**.

Locking data structures

```
// lockobject.cxx
class atomic_int {
private:
  omp_lock_t the_lock;
  int _value{0};
public:
  atomic_int() {
    omp_init_lock(&the_lock);
  };
  atomic_int( const atomic_int& )
      = delete;
  atomic_int& operator=( const atomic_int& )
      = delete;
  ~atomic_int() {
    omp_destroy_lock(&the_lock);
  };
```

Running this:

```
atomic_int my_object;
vector<std::thread> threads;
for (int ithread=0; ithread<NTHREADS; ithread++) {
  threads.push_back
```

First touch and containers

We make a template for uninitialized types:

```
// heatalloc.cxx
template<typename T>
struct uninitialized {
  uninitialized() {};
  T val;
  constexpr operator T() const {return val;};
  T operator=( const T&& v ) { val = v; return val; };
};
```

so that we can create vectors that behave normally:

```
vector<uninitialized<double>> x(N),y(N);

#pragma omp parallel for
for (int i=0; i<N; i++)
  y[i] = x[i] = 0.;
x[0] = 0; x[N-1] = 1.;
```