OpenMP case studies
Victor Eijkhout
SDS 335 2022
Case study: molecular dynamics
2. 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:

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

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

\[
\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}.
\]
Let’s start with a couple of building blocks.

```c
// 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;  
}
```
4. Sequential code in C

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

```c
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:

```c
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 $\vec{F}_{ij}$ is only computed for $j > i$, and then added to both $\vec{F}_i$ and $\vec{F}_j$. 
5. In C++

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

```cpp
// 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:

```cpp
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;
    }
}
```
6. Exercise

Is the outer loop parallelizable? The inner? Both together?
7. Solution 1: full interactions

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

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

```cpp
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
```
9. 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?)

![Graph showing speedup comparison]
10. Solution 2: atomic updates

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

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

To deal with the conflicting $j_p$ writes, we make the writes atomic:

```c
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;
}
```
What happens with one thread?
12. 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.

```c
#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
```
13. Results

![Graph showing speedup over different times with a reference line and bars at 18, 37, and 56]
14. All results together

The graph shows the speedup for different thread counts and various synchronization mechanisms. The x-axis represents the number of threads, and the y-axis represents the speedup.

- **Ideal**: The ideal speedup, which is the same as the number of threads.
- **Sequential**
- **Inner loop reduction**
- **Triangle atomic**
- **Full atomic**

The graph compares the speedup of these synchronization methods under different thread counts (1, 18, 37, 56) with a linear scale.
Search: 8 queens
15. Tree traversal

Search: traverse the tree, and abort unsuccessful branches

DFS, not BFS
16. Eight queens

```cpp
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)
    } // end for(col)
    return {};
};
```
```cpp
placement initial; initial.fill(empty);
optional<placement> eightqueens;
#pragma omp parallel
#pragma omp single
eightqueens = place_queen(0, initial);
```
18. More tasks

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

```c
#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)
} // end for
```
19. 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:

```cpp
// 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)
}
// end if(col<N)
return result;
```
This is a 1000 times slower than sequential. Why?
Body of the loop over columns:

```cxx
// 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)
```
22. Timing

![Graph showing timing results with cores used on the x-axis and sec on the y-axis.]

Still not great. Conclusion?
Parallel loops in C++ can use range-based syntax:

```cpp
// 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.
24. Iterators

Support for C++ iterators

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

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

```cpp
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:

```cpp
auto tmin = generic_reduction<float>(fdata);
```
Reduction can be applied to any class for which the reduction operator is defined as `operator+` or whichever operator the case may be.

```cpp
// 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 );
  }
};
```

```cpp
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 ??.
27. Locking data structures

```cpp
// lockobject.cxx
class object {
private:
    omp_lock_t the_lock;
    int _value{0};
public:
    object() {
        omp_init_lock(&the_lock);
    };
    ~object() {
        omp_destroy_lock(&the_lock);
    };
    int operator +=( int i ) {
        // atomic increment
        omp_set_lock(&the_lock);
        _value += (s>0); int rv = _value;
        omp_unset_lock(&the_lock);
        return rv;
    };
    auto value() { return _value; };
};
```
28. First touch and containers

We make a template for uninitialized types:

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

so that we can create vectors that behave normally:

```cpp
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.;
```