Performance obstacles in OpenMP:

- Fork/Join
  Time to create new threads, rescheduling
- Non-parallelized regions, serial sections
  Amdahl's law, Speedup < 1/s
- Synchronization
  Explicit/implicit barriers (for/sections/single)
- Load imbalance
  Trivial or naïve load balancing with OpenMP directives
- Cache misses => “communication”
  True/false sharing
- Non-optimal data placement on NUMA
  Costly remote memory accesses

Non-parallelized regions, serial sections:

* Split work at highest level => force code to be parallel.
* Overlap serial sections (ordered/single/master/critical) with other parallel activities, e.g., as using ordered above and as using single in iterative solver below.
* Have different names on different critical sections.

Synchronization:

* Minimize load imbalance.
* Analyze and remove implicit barriers between independent loops, using nowait.
* Use large parallel regions, avoid fork-join synch (also good for cache performance, threads not re-scheduled).
* Overlap activities to remove barriers (Iterative solver).
* Use locks to remove global barriers (LU-factorization).
Load imbalance:

* Use schedule-directive

```c
#pragma omp parallel for schedule(type,[chunk])
for(i=0;i<n;i++)
    work(a[i]);
```

Where: type = static, dynamic, guided  
**static**: regular work load, good cache locality  
**dynamic, guided**: irregular or unknown work load  
Large chunk is in general good (trade off with load)

* Use explicit load balancing

```c
computeload(LB,UB,nthr);
#pragma omp parallel private(i,id)
{
    id=omp_thread_num();
    for (i=LB(id);i<UB(id);i++)
        work(a[i]);
}
```

Problems with the schedule directive:

Ex 1: 13 iterations, 6 threads  
default schedule  => 3,3,3,3,1,0  
explicit partition  => 3,2,2,2,2,2

Ex 2: How to perform a 2D-decomposition?  
E.g., the Ocean modelling problem

Ex 3: Consider 2 threads and 7 tasks with  
weights (run time): 5,2,3,4,5,2,10  
Static  => 14,17  
dynamic,1  => 11,20  
bin-pack  => 16,15

=> Use explicit scheduling if bad performance
**Cache misses:** ("communication")

- Cold/compulsory - first time access
- Conflict/capacity - "full" cache
- True sharing - invalid data
- False sharing - invalid cache line

**Minimize cache misses:** (Application dependent)

- Re-use data as much as possible before replace, e.g., by cache blocking and loop fusion.
- Access data in sequence, e.g., by grouping data and by arranging loop order.
- Create dense data partitions, e.g., by using large chunk size following the data layout.

Note: schedule(static,1) generates a lot of false sharing, better with schedule(static,8) for scheduling whole cache lines.
Data placement, the NUMA problem:
(Consider multi-socket multi-core/processor nodes, e.g., AMD Magny Cores)

Non-Uniform Memory Access times, i.e., different access times to local memory close to your core and to remote memory close other cores.

=> Need control of data placement and localization of the data accesses (explicit user control!)

Memory placement often handled with first touch, i.e., memory is bound to the first touching thread with page granularity (typically 8KB).

=> Use parallel initialization with same access pattern as in the computations

(If serial init, all data allocated in touching thread's node. All other threads generate remote accesses and we get memory congestion.)

Avoid serial init on NUMA
**Note:** Need static access pattern, e.g., using schedule(dynamic) destroys the data locality. Use schedule(static) or user supplied schedule

**Data location and load balancing**

```c
loadbal(lb,ub,nthr);
#pragma omp parallel private(id,j)
{
    id=omp_thread_num();
    for (j=lb(id); j<ub(id), j++)
        A(j)=INIT(j);    // INIT, FIRST TOUCH

#pragma omp barrier
    for (j=lb(id); j<ub(id), j++)
        WORK(A(j));      // WORK, STATIC ACCESS
}
```

*Good for cache performance on a multicore node!*

---

**Case studies:**

1. **Iterative solver**

   ```
   while (norm>eps)
   y=Ax
   norm=||y-x||
   x=y
   end while
   ```

   E.g. - Jacobi for linear system of equations
   - Conjugate Gradient for optimization
   - Power method for eigenvalues
```c
#pragma omp parallel
{
    while (norm > eps)
    {
        #pragma omp for private(j)
        for i=1,n
            for j=1,n
                y(i)=y(i)+A(i,j)*x(j)
            end for
        end for

        #pragma omp single
        norm=0

        #pragma omp for reduction(+:norm)
        for i=1,n
            norm=norm+(y(i)-x(i))**2
        end for

        #pragma omp single
        swap(x,y)
    }
}
```

4 barriers per iteration

Improve:
- Make x, y private
- Remove barriers between independent loops
- Unroll 2 iterations

1 barrier per iteration

```c
norm1=0; norm2=0;
#pragma omp parallel firstprivate(x,y)
while (1)
    #pragma omp for private(j) nowait
    for i=1,n
        for j=1,n
            y(i)=y(i)+A(i,j)*x(j)
        end for
    end for

    #pragma omp single nowait
    norm2=0

    #pragma omp for reduction(+:norm1)
    for i=1,n
        norm1=norm1+(y(i)-x(i))**2
    end for

    #pragma omp single nowait
    swap(x,y)
    if (norm1 <= eps) break

    //end parallel
```
2. Sparse matrix-vector multiplication (y=Ax)

Use compressed sparse row format

- `val(nnz)` : nonzeros in matrix
- `col(nnz)` : column of nonzeros
- `row(nrow+1)` : starting pos of rows

Algorithm:

```c
register double d0;

#pragma omp for private(i,j,d0)
for ( i = 0; i < nrow; i++ )
{
    d0 = 0.0;
    for ( j = row[i]; j < row[i+1]; j++ )
        d0 += val[j] * x[col[j]];
    v[i] = d0;
}
```

What are the parallel overheads?
(Assume MxV is part of an iterative solver, e.g., Conjugate-Gradient solver)
Schedule(static):
- Load imbalance, different number of non-zeros per row
- True sharing, need updates of x-vector
- Remote accesses if large bandwidth

Schedule(dynamic):
- True sharing, need updates of x-vector
- False sharing, multiple updates of cache lines
- Remote accesses regardless of bandwidth (and bad cache utilization in accesses of x and y)

Note: Smaller bandwidth decreases true sharing remote accesses => Use bandwidth minimization, e.g., Reverse Cuthill-McKee

Real application, GEMS:
Maxwell's equations discretized with FEM-grid around a fighter jet => Ax=b with 1.8 million unknowns, solved with the CG method
Performance of GEMS solver:

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>RCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>1.24</td>
<td>1.01</td>
</tr>
<tr>
<td>Time</td>
<td>336.7</td>
<td>234.6</td>
</tr>
</tbody>
</table>

Table 1: Sun E10K, UMA

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>RCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>1.24</td>
<td>1.01</td>
</tr>
<tr>
<td>Time</td>
<td>131.3</td>
<td>74.8</td>
</tr>
<tr>
<td>L2 miss</td>
<td>427M</td>
<td>376M</td>
</tr>
<tr>
<td>Remote</td>
<td>125M</td>
<td>70M</td>
</tr>
</tbody>
</table>

Table 2: Sun Fire 15K, NUMA

3. LU-factorization

for $k=1$ to $n$
    for $i=k+1$ to $n$
        $A(i,k)=A(i,k)/A(k,k)$
        end for
    for $i=k+1$ to $n$
        for $j=k+1$ to $n$
            $A(i,j)=A(i,k)*A(k,j)$
        end for
    end for
end for

Parallelize update of $A(i,j)$ over the i-loop.

Parallel overheads:

- Frequent global synch of all threads (for each $k$)
- Non-static data partitions (the parallel loops shrink) lose data locality

Improvements:

- One large parallel region (including k-loop)
- Static partitioning cyclicly over columns
- First touch using parallel initialization
- Individual synchronization using locks
|-- Set up locks for each column
do i=1,n
   call omp_init_lock(lck(i))
end do

!$OMP PARALLEL PRIVATE(i,j,k,thrid)
   thrid=omp_get_thread_num();
!$OMP END PARALLEL

|-- Initiate (parallel first touch)
!$OMP DO SCHEDULE(STATIC,chunk)
do j=1,n
   do i=1,n
      A(i,j)=1.0/(i+j)
   end do
   call omp_set_lock(lck(j))
end do
!$OMP END DO

|-- First column of L
if (thrid==0) then
   do i=2,n
      A(i,1)=A(i,1)/A(1,1)
   end do
   call omp_unset_lock(lck(1))
end if

|-- LU-factorization
do k=1,n
   call omp_set_lock(lck(k))
   call omp_unset_lock(lck(k))
!$OMP DO SCHEDULE(STATIC,chunk)
do j=1,n
   if (j>k) then
      do i=k+1,n
         A(i,j)=A(i,j)-A(i,k)*A(k,j)
      end do
      if (j==k+1) then
         do i=k+2,n
            A(i,k+1)=A(i,k+1)/A(k+1,k+1)
         end do
         call omp_unset_lock(lck(k+1))
      end if
   end if
end do
!$OMP END DO NOWAIT
end do
!$OMP END PARALLEL

Performance: (Sun E10K)

<table>
<thead>
<tr>
<th>Threads</th>
<th>LU-standard</th>
<th>LU-lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31.4</td>
<td>29.7</td>
</tr>
<tr>
<td>2</td>
<td>5.83</td>
<td>3.32</td>
</tr>
<tr>
<td>4</td>
<td>3.44</td>
<td>1.69</td>
</tr>
<tr>
<td>8</td>
<td>2.37</td>
<td>0.97</td>
</tr>
<tr>
<td>16</td>
<td>2.62</td>
<td>0.63</td>
</tr>
<tr>
<td>24</td>
<td>3.20</td>
<td>0.44</td>
</tr>
</tbody>
</table>

⇒ 6 times performance improvement! Also, superlinear speedup due to better cache performance (data fits)

Note: similarity with Gram-Schmidt and Pthreads