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

```
#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

```
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

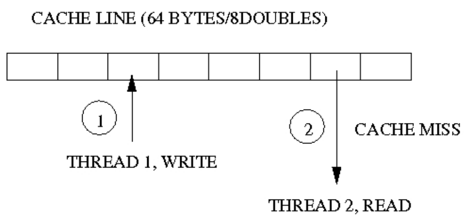
Ex 3: Consider 2 threads and 7 tasks with
weights (run time): 5,2,3,4,5,2,10

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



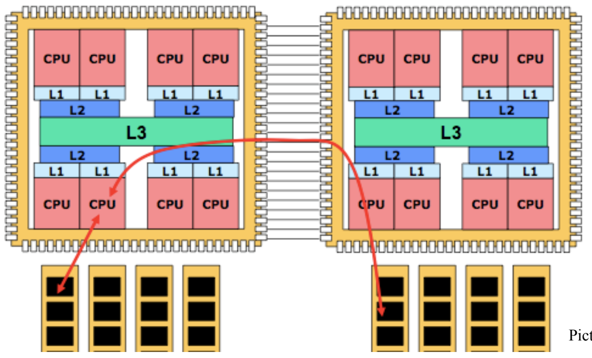
Minimize cache misses: (Application dependent)

* 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 multicore nodes, e.g., AMD Magny Cores)



Picture by Erik Hagersten

Non-**U**niform **M**emory **A**ccess 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 initalization 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.)

```
 ! Init
 do i = 1, N
    do j = 1, M
       arrays(i,j) = ....
    end do
 end do

!$OMP PARALLEL SHARED(arrays)
    .
    .
!$OMP DO SCHEDULE(STATIC)
 do i = 1, N
    do j = 1, M
       arrays(i,j) = ....
    end do
 end do
    .
    .
```

Avoid serial init on NUMA

Note: Need static access pattern, e.g., using schedule(dynamic) destroys the data locality

Remedy: use user supplied load balancing

```
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

```
!$omp parallel
    do while(norm>eps)

        !$omp do private(j)
        do i=1,n
            do j=1,n
                y(i)=y(i)+A(i,j)*x(j)
            end do
        end do
        !$omp end do

        !$omp single
        norm=0
        !$omp end single

        !$omp do reduction(+:norm)
        do i=1,n
            norm=norm+(y(i)-x(i))**2
        end do

        !$omp single
        swap(x,y)
        !$omp end single

    end do
!$omp end parallel
```

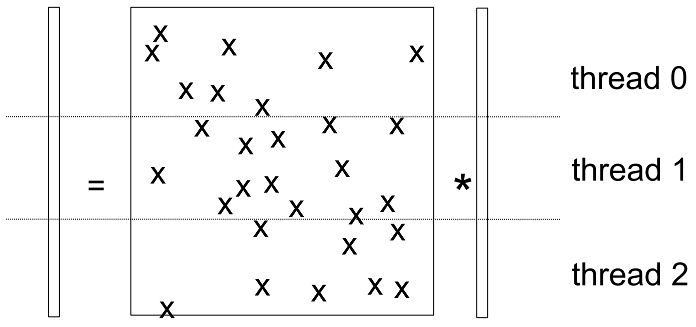
=> 4 barriers per iteration

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

=> 1 barrier per iteration

```
norm1=0; norm2=0;
!$omp parallel firstprivate(x,y)
do while (1)

    !$omp do private(j)
    do i=1,n
        do j=1,n
            y(i)=y(i)+A(i,j)*x(j)
        end do
    end do
    !$omp end do nowait

    !$omp single
    norm2=0
    !$omp end single nowait

    !$omp do reduction(+:norm1)
    do i=1,n
        norm1=norm1+(y(i)-x(i))**2
    end do

    swap(x,y)
    if (norm1<=eps) break

    !$omp do private(j)
    do i=1,n
        do j=1,n
            y(i)=y(i)+A(i,j)*x(j)
        end do
    end do
    !$omp end do nowait

    !$omp single
    norm1=0
    !$omp end single nowait

    !$omp do reduction(+:norm2)
    do i=1,n
        norm2=norm2+(y(i)-x(i))**2
    end do

    swap(x,y)
    if (norm2<=eps) break

end do
!$omp 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(nrows)	: starting pos of rows

Algorithm:

```
!$omp parallel do private(d,j)
do i=1,nrows
    d=0
    do j=row(i),row(i+1)-1
        d=d+val(j)*x(col(j))
    end do
    y(i)=d
end do
```

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

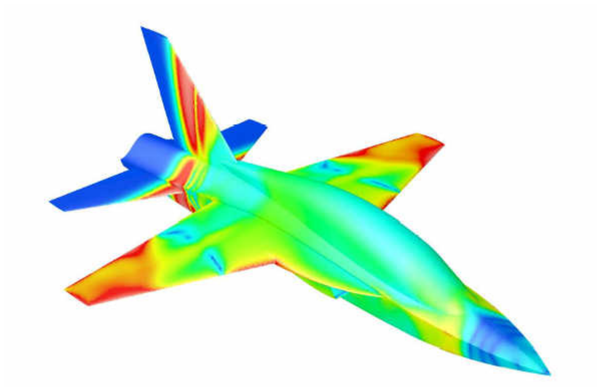
Schedule(dynamic):

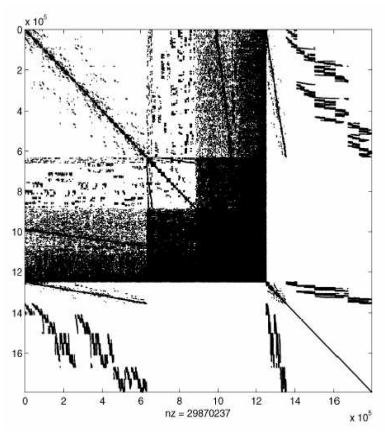
- True sharing, need updates of x-vector
- False sharing, multiple updates of cache lines
- Remote accesses regardless of bandwidth (and bad cache utilization in accesses of x and y)

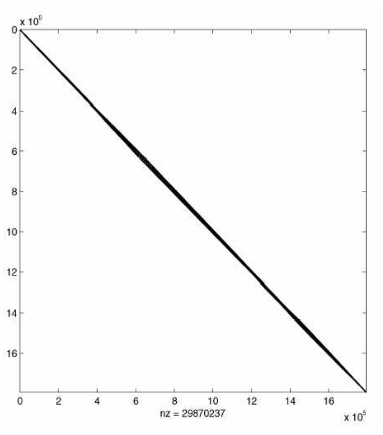
Note: Smaller bandwidth decreases true sharing remote accesses => Use bandwidth minimization, e.g., Reverse Cuthill-McKee

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







Original matrix

Bandwidth minimized

Performance of GEMS solver:

	Original	RCM
Load	1.24	1.01
Time	336.7	234.6

S=1.44

Table 1: Sun E10K, UMA

	Original	RCM
Load	1.24	1.01
Time	131.3	74.8
L2 miss	427M	376M
Remote	125M	70M

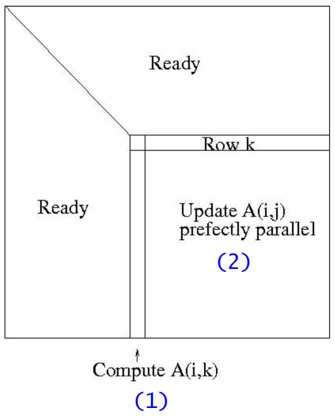
S=1.76

Table 2: Sun Fire 15K, NUMA

[Ref: H. Löf, J. Rantakokko, *Algorithmic Optimization of a Conjugate Gradient Solver on Shared memory systems*, International Journal of Parallel, Emergent and Distributed Systems, Vol 21, 2006.]

3. LU-factorization (Lab 3, task 6)

```
for k=1 to n
    for i=k+1 to n
(1)     A(i,k)=A(i,k)/A(k,k)
    end for
    for i=k+1 to n
        for j=k+1 to n
(2)         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();

!– Initate (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
```

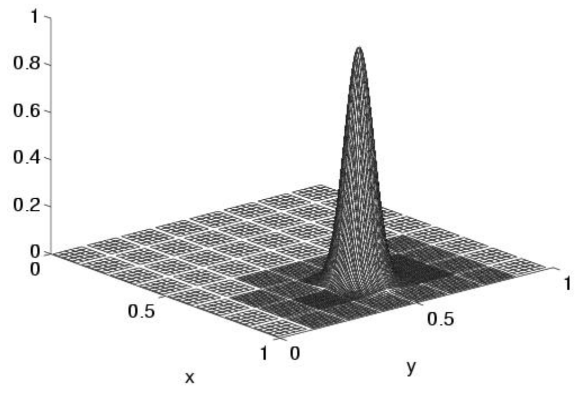
Performance: (Sun E10K)

Threads	LU-standard	LU-lock
1	31.4	29.7
2	5.83	3.32
4	3.44	1.69
8	2.37	0.97
16	2.62	0.63
24	3.20	0.44

=> 6 times performance improvement!
What about multi-core? Experiment at hands-on session.

4. Adaptive Mesh Refinement

(Research example of the data placement problem on NUMA)



Use higher resolution (block wise) in areas of interest

The problem:

- Pulse moves in the domain => The grid adaption is dynamic and follows the pulse.
- We do the parallelization over the blocks, i.e., each thread is responsible for a number of blocks.

↓

Work load per thread change as the blocks are refined or coarsened => Need to repartition data at runtime (even blocks that are not changed).

How? What do we need to consider?

- Optimize load balance, equal work load
- Minimize communication, i.e., neighbor blocks within the same partition as far as possible
- Minimize change of ownership for blocks

=> Use a *diffusion algorithm*, e.g., from Jostle or ParMetis.

Problem: Data locality is destroyed!

Remedy: `Migrate-on-next-touch*`

(* Sun specific solution, re-do first touch. On other systems, re-allocate blocks on new owners and do first touch.)

Performance results:

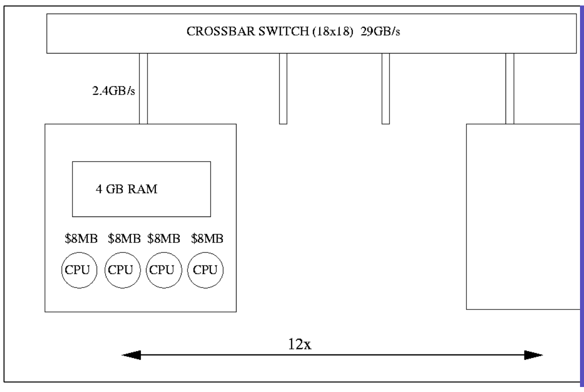
	No migration	Migration
CPU time	6.64h	3.99h
Remote acc	62.9%	8.1%

Performance: SunFire 15K

Data partitioning and explicit control of data => 66% performance improvement (2.6h)

[Ref: M. Norden, H. Löf, J. Rantakokko, S. Holmgren, *Dynamic data migration for structured AMR solvers*, International Journal of Parallel Programming, 2007]

SunFire 15K (2002), “Ngorongoro”



- 12x4 Ultra Sparc IIIcu, 900MHz + 1x4 US IV+ dual-core, 1.3GHz
- 12x4 GB (48GB RAM)
- Peak performance: 2x48x900MFlops = 86.4 Gflop/s
- List price: ~20M SEK