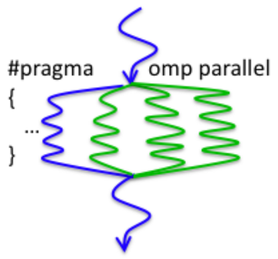
Shared Memory (Multi-Core) Programming with OpenMP

Jarmo Rantakokko
Senior lecturer, IT UU



OpenMP: Open specification for Multi Processing

(www.openmp.org, v1.0 1997 - v4.5 2015, we will use v3.0)

Shared address space model, based on *threads*

Thread: - Light weight process, global addresses
- Private program counter, independent
- Private stack pointer, private data

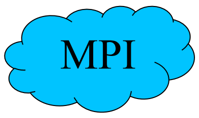
⇒ All threads have access to global data, can run in parallel and have some private data on stack.

On a multi-core node the threads are scheduled over the CPU's to the different cores.
⇒ One node on IT-servers can run 8 parallel threads.

Insert compiler directives for parallelization of Computations ⇨ high-level model

```
#pragma omp parallel for
for (i=2;i<=N-1;i++)
    A[i]=F(B[i-1]+B[i]+B[i+1]);
```

Loop is automatically parallelized over all threads, different iterations on different theads. Arrays A and B are global data, loop variable i is private.



```
NLOC=N/NPROC
ALLOCATE (A(NLOC),B(NLOC))
. . .
(Standard send/recv avoiding deadlock)

IF (MOD(PID,2)==1) THEN
   CALL MPI_SEND(B(1),LEFT)
   CALL MPI_RECV(TEMP1,LEFT)
ELSEIF (MOD(PID,2)==0 AND PID<NPROC-1)
   CALL MPI_RECV(TEMP2,RIGHT)
   CALL MPI_SEND(B(NLOC),RIGHT)
END IF
IF (MOD(PID,2)==1 AND PID<NPROC-1) THEN
   CALL MPI_SEND(B(NLOC),RIGHT)
   CALL MPI_RECV(TEMP2,RIGHT)
ELSEIF (MOD(PID,2)==0 AND PID>0)
   CALL MPI_RECV(TEMP1,LEFT)
   CALL MPI_SEND(B(1),LEFT)
END IF
                (Simpler with non-blocking communication
                 MPI_Irecv followed by MPI_Send)
IF (PID>0) THEN
   A(1)=F(TEMP1+B(1)+B(2))
END IF
FOR (I=2; I<NLOC-1; I++)
   A(I)=F(B(I-1)+B(I)+B(I+1))
END DO
IF (PID<NPROC-1) THEN
   A(NLOC)=F(B(NLOC-1)+B(NLOC)+TEMP2)
END IF
```

Pthreads

```
 struct thread_data
{
        int j1;
        int j2;
};
void *compute(void *arg)
{
        int j1,j2;
        struct thread_data *index;
        index=(struct thread_data *)arg;
        j1=index->j1;
        j2=index->j2;

        for (j=j1;j<j2;j++)
             A[i]=F(B[i-1]+B[i]+B[i+1]);
}
int main(){
…
for(t=0; t<NUM_THREADS; t++) {
         index[t].j1=t*len/NUM_THREADS;
         index[t].j2=(t+1)*len/NUM_THREADS;
         pthread_create(&threads[t], &attr, compute,
                        (void *) &index[t]); }
```

OpenMP directives:

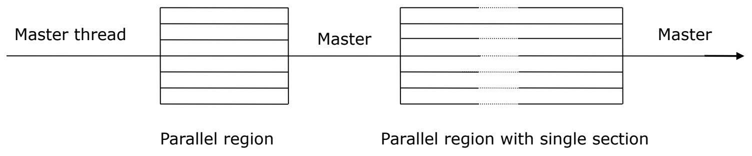
Parallel (main, fork threads)

Data sharing
- shared
- private
- firstprivate
- lastprivate
- threadprivate

Work sharing
- for
  - reduction
  - schedule
    - static, dynamic, guided
  - ordered
  - collapse
- sections
- tasks

Serial sections
- single
- master
- critical
- atomic
- ordered

Synchronization
- barrier
- flush
- nowait



Parallel region

Parallel region with single section

OpenMP library functions:

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num
- omp_set_nested
- and more (e.g. lock)

Allows for more flexible and user controlled (e.g. load balancing) programming than with the standard directives.

Environment variables: (export VARIABLE=value)

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_NESTED
- and more (stacksize, wait policy)

To run on 4 threads, before start of program do:

```
export OMP_NUM_THREADS=4
```

Directives: (Support only in Fortran/C/C++)

C/C++: #pragma omp *directive*
{ code block }

Fortran: !$omp *directive*
code block
!$omp end *directive*

Note: The directives are ignored by non-supporting compiler or if OpenMP-flag is turned off in compiling.
⇒Portable code between single CPU, multi-core, and general parallel computers.

Also, possible to parallelize code incrementally (start with heaviest routine and continue until sufficient parallelism and performance are achieved)

Parallel: (Fork-Join of threads)

```
#pragma omp parallel [subdirectives]
{
"parallel code"
}
```

Subdirectives:

if (true/false)	-- parallel/serial
num_threads(int)	-- Number of threads
reduction (op:var)	-- parallel reduction

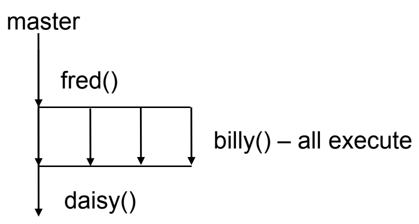
+ directives for data sharing (private/shared)

Parallel: (Fork-Join of threads)

```
#pragma omp parallel
{
"parallel code"
}
```

If no subdirectives, all data shared (global) and all code executed in parallel by all threads. At the end of parallel the threads are synchronized and joined.

```
Ex: program p1
    ...
    call fred()
    #pragma omp parallel
    { call billy() }
    call daisy()
```



Example HelloWorld:

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

#pragma omp parallel
{
   printf("Hello World! %d\n”,omp_get_thread_num());
}

}
```

Task: Compile and run the program helloworld.c

- gcc –fopenmp helloworld.c –o hello
- ./hello

Run on different number of threads.
In what ways can we change the number of threads?
What is the default number of threads?

Data sharing:

- **shared**([list of variables]) - default
- **private**([list of variables])

Ex: program p2

```
...
a=100; b=0;
#pragma omp parallel private(a) num_threads(10)
{
    b=b+1000;
    a=b+a;
 }
printf(“a= %d, b= %d \n”, a,b);
```

Task: What is the result, run the program datasharing.c several times and explain the output.

Note: All private variables are allocated on the stack
=> uninitialized at entry and removed at exit,
original a not equal to *private a*!

Note2: Shared variables must be protected from
simultaneous writes by different threads!
(Use critical directive or locks.)

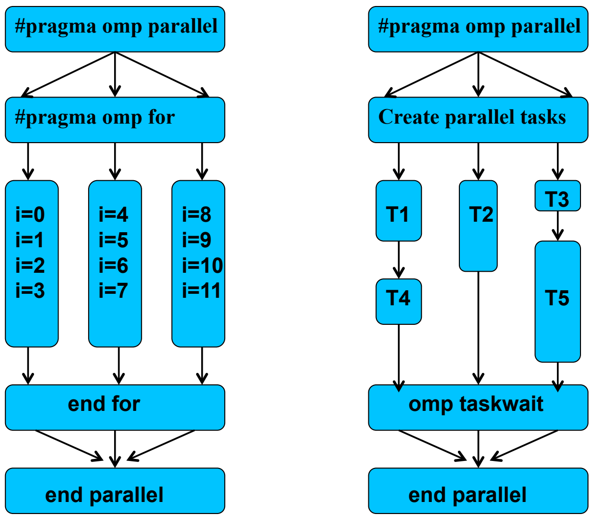
- **firstprivate**([list of variables])
  As private but the variables are initialized from the original variable (in master) before parallel.

- **lastprivate**([list of variables])
  At exit, the original variable gets the value from the thread executing the last iteration in a loop using the for-directive or the last section in the sections-directive.

- **threadprivate**([list of variables])
  Make global file scope variables local and persistent to a thread through the execution of multiple parallel regions.

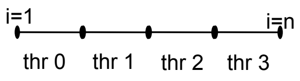
Work sharing (within parallel)

- Loop level parallelism – **for**
- Task parallelism – **sections, tasks**



for-directive:

```
#pragma omp for [subdirectives]
for (i=1; i<=n; i++)
   { loop-body }
```



Subdirectives:

- Private
- Firstprivate
- Lastprivate
- Reduction
- Schedule
- Ordered
- Collapse

Without subdirectives, loop counter is private, loop space is divided statically into nthr equal pieces, and run in parallel (different iterations in different threads). Threads are synchronized at end of the for-directive.

Note: We must have a perfectly parallel loop!

Example: Enumeration sort

```
for (j=0;j<len;j++)
{
  rank=0;
  for (i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;
  outdata[rank]=indata[j];
}
```

For each element (j) check how many other elements (i) are smaller than it => rank
Perfectly parallel tasks for each element (j)

Task: Parallelize the j-loop in enumsort.c and set appropriate variables as private.
What speedup can you get for 50,000 elements?

Reduction(op:[list of variables])

Performs a global reduction using
op=+,-,*,max,min, or a logical operator

```
sum=0;
#pragma omp parallel for reduction(+:sum)
 for (i=0;i<n;i++) sum=sum+a[i];
```

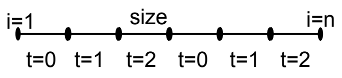
```
sum=0;
#pragma omp parallel private(locsum)
 {
      locsum=0;
      #pragma omp for
      for (i=0;i<n;i++) locsum=locsum+a[i];
      #pragma omp critical
    { sum=sum+locsum; }
 }
```

Task: Parallelize the inner i-loop in enumsort.c and compare the performance with your first parallelization.
What are the performance obstacles and/or advantages?

Schedule(type, [size])

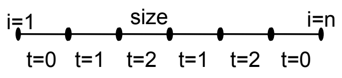
Divides the iteration space into chunks=size and schedules the chunks to threads according to type. (size=n/nthr by default)

type=static:



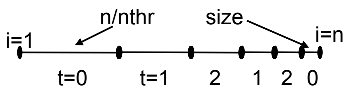
Assign the chunks cyclicly to threads

type=dynamic:



Dynamic scheduling, as soon as a thread is ready it gets a new chunk

type=guided:



As dynamic but the chunk size is decreasing towards end. Minimizes synchronization time.

type=runtime:

Decide at runtime using the environment variable export *schedule=type* (where *type* is some above).

type=auto:

Let the run-time system and/or compiler decide automatically.

Note: Static scheduling is good for data locality (cache) while dynamic/guided good for load balance.

Task: Try different scheduling options in the program loop.c, what gives the best performance, what is the theoretically minimal runtime, how can we get that?

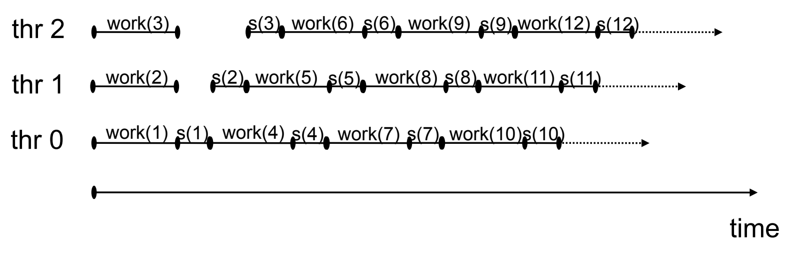
Ordered

Only one thread is allowed to the ordered block at a time and sequentially in loop order. Useful for I/O.

```
#pragma omp parallel
{
    #pragma omp for schedule(static,1) ordered
    for (i=0;i<n;i++)
    {
        call work(i)            ! parallel work

        #pragma omp ordered
        { call s(i) }           ! serial section
    }
}
```

Assume work(i)>>s(i) => Parallelism, pipelining effect

Static,1:



What happens if we use default scheduling (size=n/nthr)?

Collapse directive

Allow *collapsing* of perfectly nested loops, i.e., form a single loop and then parallelize that. Example: parallelize both i and j-loop in MxM

```
#pragma omp parallel
{
#pragma omp for collapse(2) private(i,j,k)
for (i=0;i<len;i++)
   for (j=0;j<len;j++)
    {
      c[i*len+j]=0.0;
      for (k=0;k<len;k++)
         c[i*len+j]+=a[i*len+k]*b[k*len+j];
    }
}
```

Task parallelism (static predefined tasks)

Sections

```
#pragma omp sections [subdirectives]
{
    #pragma omp section
        { task 1 }

    #pragma omp section
        { task 2 }

    etc.
}
```

Subdirectives:

- Private
- Firstprivate
- Lastprivate
- Reduction

The sections/tasks are scheduled (statically) to the threads and run in parallel. At end of sections the threads are synchronized. (No load balancing).

Nested parallelism (load balancing of sections)

```
omp_set_nested(1);
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    {
    #pragma omp parallel for num_threads(P1)
   for (k=0;k<n1;k++)
       call WORK1(A[K])
    }
    #pragma omp section
    {
    #pragma omp parallel for num_threads(P2)
   for (k=0;k<n2;k++)
       call WORK2(A[K])
    }

}
```

Assign appropriate number of threads to each section.

Task: Make a two-level parallelization of enumsort.c

Task directive (dynamic tasks)

Can implement task-queues that are scheduled dynamically to all available threads in a parallel environment. Tasks can be generated at run-time.

Generate a task:

```
#pragma omp task [if/untied/'datasharing']
{ task }
```

Wait for all tasks to complete

```
#pragma omp taskwait
```

Task scheduling points at following locations:

1. Generation of task
2. Last instruction in task
3. Taskwait-directive
4. Implicit and explicit barriers

Task: Study and run the program task.c. What is the effect of if and nowait?

Serial sections

Avoid terminating threads, lose data in cache if threads rescheduled to different CPUs or cores (with fork-join model).

#pragma omp single [subdirectives]
The code-block within single is executed only by one thread, the others skip and wait at the end of block.
Subdirectives: - private
- firstprivate

#pragma omp master
The code-block is executed only by master thread, the other skip and continue (no barrier).

#pragma omp critical [name]
The code-block is executed by one thread at a time.
As ordered but no predefined order.
If no name all critical sections have the same name.
Only one critical section with the same name can be executed by one thread at a time.

#pragma omp atomic
Atomic update by one thread at a time. As critical but applies only for a one line expression. (Does not include a memory flush.)

Synchronization

Done implicitly at end of:
- parallel
- for
- sections
- single

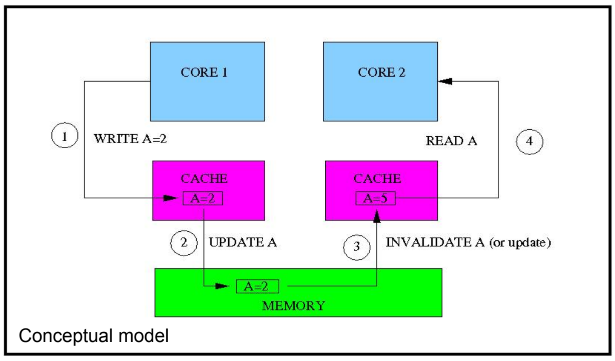
Explicit barrier:
#pragma omp barrier
(#pragma omp taskwait)

Can override with *nowait:*

```
#pragma omp for nowait
for (i=0;i<n;i++)
    { code }
```

Note: If *nowait* be careful not to use data updated by other threads, *nowait* overrides memory flush!

In a **memory flush** all thread visible shared variables are refreshed (caches invalidated and memory updated). A memory flush is performed at all barriers (implicit and explicit) and before/after a critical directive.



=> *Be very careful with nowait !!! (Nowait removes 2 & 3)*

OpenMP has a *relaxed-consistency* model, i.e., the threads can cache data not keeping exact consistency.

Task: Run the program memory.c and explain its output. How can we fix the problem?

```
int main(int argc, char *argv[]) {
int id,nthr;

 #pragma omp parallel private(id)
   {
    nthr=-1;
    id=omp_get_thread_num();

    #pragma omp single
    { nthr=omp_get_num_threads(); }

    printf("Hello World! %d %d\n",id,nthr);
   }
}
```

Synchronization with locks

Can lock a code section and/or data only accessible to a specific thread. Routines include a flush.

```
omp_init_lock - omp_destroy_lock
omp_set_lock  - omp_unset_lock
omp_test_lock
```

Example:

```
omp_lock_t lockvar;
omp_init_lock(lockvar);
…
#pragma omp parallel {
…
   omp_set_lock(lockvar);
   sum=sum+a;
   omp_unset_lock(lockvar);
}
```

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

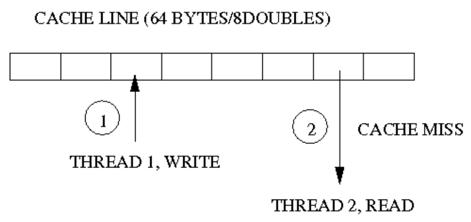
Ex 3: Consider 2 threads and 7 tasks with
weights (run time): 5,2,3,4,5,2,10

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



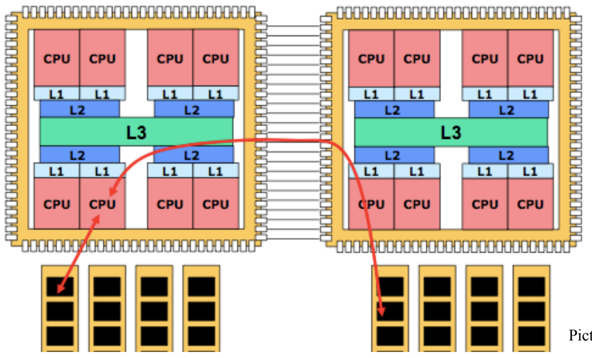
Minimize cache misses: (Application dependent)

* 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

```
#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)

    end while
}
```

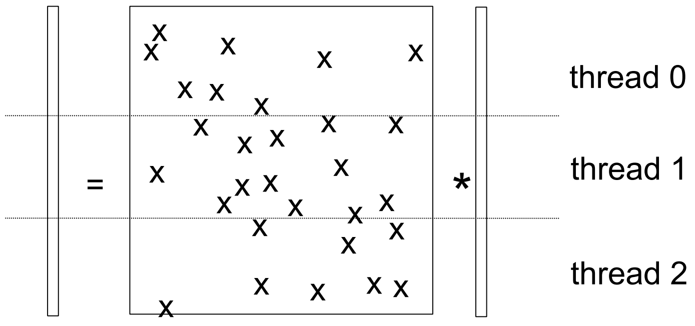
=> 4 barriers per iteration

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

=> 1 barrier per iteration

```
norm1=0; norm2=0;
#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

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

    #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
    norm1=0

    #pragma omp for reduction(+:norm2)
    for i=1,n
        norm2=norm2+(y(i)-x(i))**2
    end for

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

end while
//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:

```
register double d0;

#pragma omp for private(i,j,d0)
  for( i = 0; i < *nrows; i++ )
   {
     d0 = 0.0;
  /*Look up and add non-zeros for row i */
     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

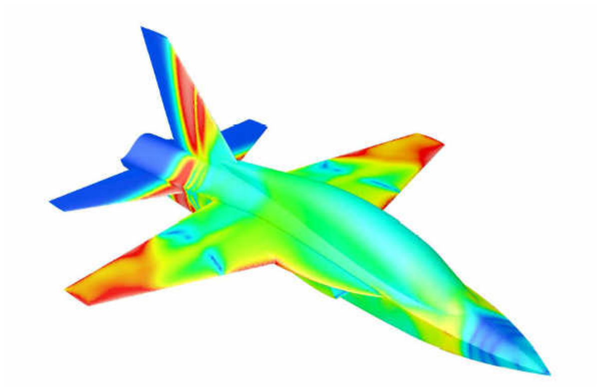
Schedule(dynamic):

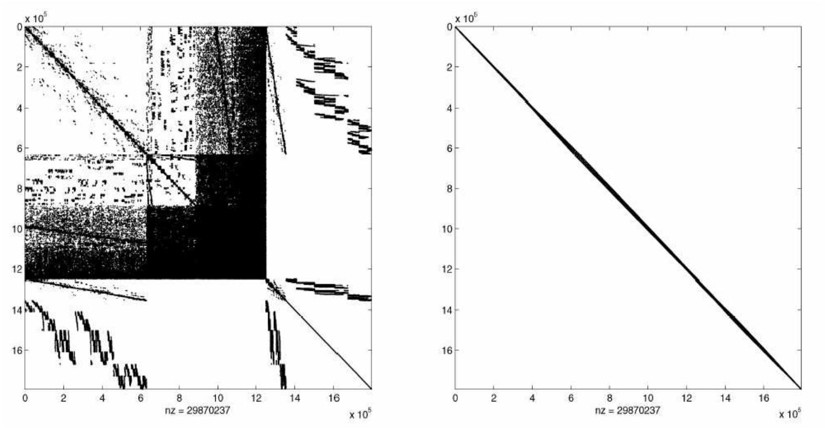
- True sharing, need updates of x-vector
- False sharing, multiple updates of cache lines
- Remote accesses regardless of bandwidth (and bad cache utilization in accesses of x and y)

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





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

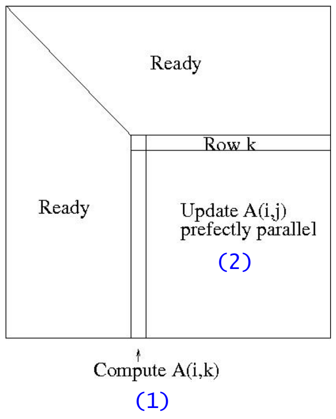
S=1.76

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.