OpenMP - shared memory parallelism

Thomas Ericsson
Computational Mathematics
Chalmers University of Technology
and the University of Gothenburg
thomas@chalmers.se

PDC Summer School 2010

Contents

OpenMP - shared memory parallelism

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.

Fortran version 1.0, Oct 1997, ver. 2.0 Nov. 2000.
C/C++ ver. 1.0 Oct. 1998, ver. 2.0 Mar. 2002.

Version 2.5 May 2005, combines the Fortran and C/C++ specifications into a single one and fixes inconsistencies.

Version 3.0, May 2008, not supported by all compilers (supported by `ifort`/`icc` ver 11.0, for example).

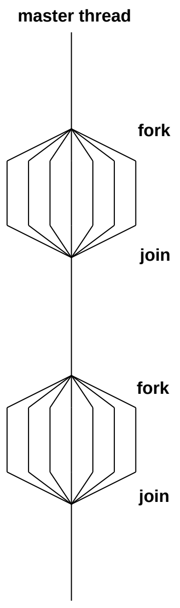
v2.5 mainly supports data parallelism (SIMD), all threads perform the same operations but on different data. In v3.0 there is better support for "tasks", different threads perform different operations (so-called function parallelism, or task parallelism).

Specifications (in PDF): `www.openmp.org`
Good readability to be standards.

For a few books look at:
`http://openmp.org/wp/resources/#Books`

The basic idea - fork-join programming model

```
program test

... serial code ...

!$OMP parallel shared(A, n)

... code run i parallel ...

!$OMP end parallel


... serial code ...

!$OMP parallel do shared(b) private(x)

... code run i parallel ...

!$OMP end parallel do


... serial code ...
```



- when reaching a parallel part the master thread (original process) creates a team of threads and it becomes the master of the team
- the team executes concurrently on different parts of the loop (parallel construct)
- upon completion of the parallel construct, the threads in the team synchronise at an implicit barrier, and only the master thread continues execution
- the number of threads in the team is controlled by environment variables and/or library calls, e.g.
  `setenv OMP_NUM_THREADS 7`
  `call omp_set_num_threads(5)` (overrides)
- the code executed by a thread must not depend on the result produced by a different thread

So what is a thread?

A thread originates from a process and is a part of that process. The threads (belonging to the particular process) share global variables, files, code, PID etc. but they have their individual stacks and program counters.

Note that we have several processes in MPI.

Since all the threads can access the shared data (a matrix say) it is easy to write code so that threads can work on different parts of the matrix in parallel.

It is possible to use threads directly but we will use the OpenMP-directives. The directives are analysed by a compiler or preprocessor which produces the threaded code.

MPI versus OpenMP

Parallelising using distributed memory (MPI):

- Requires large grain parallelism to be efficient (process based).
- Large rewrites of the code often necessary
  difficult with "dusty decks".
  May end up with parallel and non-parallel versions.
- Domain decomposition; indexing relative to the blocks.
- Requires global understanding of the code.
- Hard to debug.
- Runs on most types of computers.

Using shared memory (OpenMP)

- Can utilise parallelism on loop level (thread based).
  Harder on subroutine level, resembles MPI-programming.
- Minor changes to the code necessary. A detailed knowledge of the code not necessary. Only one version.
  Can parallelise using simple directives in the code.
- No partitioning of the data.
- Less hard to debug.
- Not so portable; requires a shared memory computer (but common with multi-core computers).
- Less control over the "hidden" message passing and memory allocation.

A simple example

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

int main()
{
  int             i, i_am, n = 10000;
  double          a[n], b[n], c[n];

  for (i = 0; i < n; i++)
    c[i] = 1.242;

// a parallel for loop
#pragma omp parallel for private(i) shared(a, b, c)
  for (i = 0; i < n; i++) {
    b[i] = 0.5 * (i + 1);
    a[i] = 1.23 * b[i] + 3.45 * c[i];
  }
  printf("%f, %f\n", a[0], a[n - 1]); // the master

// a parallel region
#pragma omp parallel private(i_am)
  {
    i_am = omp_get_thread_num();  // 0 to #threads - 1
    printf("i_am = %d\n", i_am);  // all threads print

    #pragma omp master
    {
     printf("num threads = %d\n",omp_get_num_threads())
     printf("max threads = %d\n",omp_get_max_threads())
     printf("max cpus    = %d\n",omp_get_num_procs());
    }  // use { } for begin/end
  }
  return 0;
}
```

Use **shared** when:

- a variable is not modified in the loop or
- when it is an array in which each iteration of the loop accesses a different element

All variables except the loop-iteration variable are **shared** by default. To turn off the default, use **default(none)**.

Suppose we are using four threads. The first thread may work on the first 2500 iterations (**n = 10000**), the next thread on the next group of 2500 iterations etc.

At the end of the parallel for, the threads join and they synchronise at an implicit barrier.

Output from several threads may be interleaved.

To avoid multiple prints we ask the master thread (thread zero) to print. The following numbers are printed:
number of executing threads, maximum number of threads that can be created (can be changed by setting **OMP_NUM_THREADS** or by calling **omp_set_num_threads**) and available number of processors (cpus).

```
ferlin > icc -openmp omp1.c
ferlin > setenv OMP_NUM_THREADS 1
ferlin > a.out
4.899900, 6154.284900
i_am = 0
num threads = 1
max threads = 1
max cpus    = 8

ferlin > setenv OMP_NUM_THREADS 4
ferlin > a.out
4.899900, 6154.284900
i_am = 3
i_am = 0
num threads = 4
max threads = 4
max cpus    = 8
i_am = 2
i_am = 1

ferlin > setenv OMP_NUM_THREADS 9
ferlin > a.out
4.899900, 6154.284900

etc.
```

On some some systems (# of threads > # of cpus = 8):

```
Warning: MP_SET_NUMTHREADS greater than available cpus
```

Make no assumptions about the order of execution between threads. Output from several threads may be interleaved.

Intel compilers: **ifort -openmp ...**, **icc -openmp ...**
GNU: **gfortran -fopenmp ...**, **gcc -fopenmp ...**
Portland group: **pgf90 -mp...**, **pgcc -mp ...**

The same program in Fortran

```
program example
  use omp_lib   ! or include "omp_lib.h"
               ! or something non-standard
  implicit none
  integer                         :: i, i_am
  integer, parameter              :: n = 10000
  double precision, dimension(n) :: a, b, c

  c = 1.242d0
!$omp parallel do private(i), shared(a, b, c)
  do i = 1, n
    b(i) = 0.5d0 * i
    a(i) = 1.23d0 * b(i) + 3.45d0 * c(i)
  end do
!$omp end parallel do   ! not necessary

  print*, a(1), a(n)    ! only the master

!$omp parallel private(i_am)  ! a parallel region
  i_am = omp_get_thread_num() ! 0, ..., #threads - 1
  print*, 'i_am = ', i_am

  !$omp master
    print*, 'num threads = ', omp_get_num_threads()
    print*, 'max threads = ', omp_get_max_threads()
    print*, 'max cpus    = ', omp_get_num_procs()
  !$omp end master

!$omp end parallel

end program example
```

!$omp or **!$OMP**. See the standard for Fortran77.
!$omp end ... instead of **}**.

Things one should not do

First a silly example:

```
...
  int a, i;

#pragma omp parallel for private(i) shared(a)
  for (i = 0; i < 1000; i++) {
    a = i;
  }
  printf("%d\n", a);
...
```

Will give you different values **999**, **874** etc.

Now for a less silly one:

```
  int i, n = 12, a[n], b[n];

  for (i = 0; i < n; i++) {
    a[i] = 1; b[i] = 2;        // Init.
  }

#pragma omp parallel for private(i) shared(a, b)
  for (i = 0; i < n - 1; i++) {
    a[i + 1] = a[i] + b[i];
  }

  for (i = 0; i < n; i++)
    printf("%d ", a[i]);       // Print results.
  printf("\n");

A few runs:
 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23  one thread
 1, 3, 5, 7, 9, 11, 13,  3,  5,  7,  9, 11  four
 1, 3, 5, 7, 9, 11, 13, 15, 17, 19,  3,  5  four
 1, 3, 5, 7, 9, 11, 13,  3,  5,  7,  3,  5  four
```

Why?

```
    thread           computation
      0         a[1]  = a[0] + b[0]
      0         a[2]  = a[1] + b[1]
      0         a[3]  = a[2] + b[2]    <--|
                                          |  Problem
      1         a[4]  = a[3] + b[3]    <--|
      1         a[5]  = a[4] + b[4]
      1         a[6]  = a[5] + b[5]    <--|
                                          |  Problem
      2         a[7]  = a[6] + b[6]    <--|
      2         a[8]  = a[7] + b[7]
      2         a[9]  = a[8] + b[8]    <--|
                                          |  Problem
      3         a[10] = a[9]  + b[9]   <--|
      3         a[11] = a[10] + b[10]
```

We have a data dependency between iterations, causing a so-called race condition.

Can "fix" the problem:

```
// Yes, you need ordered in both places

#pragma omp parallel for private(i) shared(a,b) ordered
  for (i = 0; i < n - 1; i++) {
  #pragma omp ordered
    a[i + 1] = a[i] + b[i];
  }
```

but in this case the threads do not run in parallel. Adding
`printf("%3d %3d\n", i, omp_get_thread_num());`
in the loop produces the printout:

```
  0   0
  1   0
  2   0
  3   1
  4   1
  5   1
  6   2
  7   2
  8   2
  9   3
 10   3
1 3 5 7 9 11 13 15 17 19 21 23
```

It is illegal to jump out from a parallel loop.
The following for-loop in C is illegal:

```
#pragma omp parallel for private(k, s)
  for(k = 0; s <= 10; k++) {
    ...
  }
```

It must be the same variable occurring in all three parts of the loop. More general types of loops are illegal as well, such as

```
  for(;;) {
  }
```

which has no loop variable. In Fortran, `do-while` loops are not allowed. See the standard for details.

Not all compilers provide warnings. Here a Fortran-loop with a jump.

```
program jump
  implicit none
  integer                :: k, b
  integer, parameter     :: n = 6
  integer, dimension(n) :: a

  a = (/ 1, 2, 3, 4, 5, 6 /)
  b = 1

 !$omp parallel do private(k) shared(a)
   do k = 1, n
     a(k) = a(k) + 1
     if ( a(k) > 3 ) exit  ! illegal
   end do

  print*, a
end program jump

% ifort -openmp jump.f90
fortcom: Error: jump.f90, line 13: A RETURN, EXIT or
         CYCLE statement is not legal in a DO loop
         associated with a parallel directive.
     if ( a(k) > 3 ) exit  ! illegal
--------------------^
compilation aborted for jump.f90 (code 1)

% pgf90 -mp jump.f90    the Portland group compiler
% setenv OMP_NUM_THREADS 1
% a.out
    2    3    4    4    5    6
% setenv OMP_NUM_THREADS 2
% a.out
    2    3    4    5    5    6
```

firstprivate variables

When a thread gets a private variable it is not initialised. Using `firstprivate` each thread gets an initialised copy.

In this example we use two threads:

```
...
  int i, v[] = {1, 2, 3, 4, 5};

#pragma omp parallel for private(i) private(v)
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  printf("\n");

#pragma omp parallel for private(i) firstprivate(v)
  for (i = 0; i < 5; i++)
    printf("%d ", v[i]);
  printf("\n");
...

% a.out
40928 10950 151804059 0 0
1 2 4 5 3   (using several threads)
```

Load balancing

We should balance the load (execution time) so that threads finish their job at roughly the same time.
There are three different ways to divide the iterations between threads, **`static`**, **`dynamic`** and **`guided`**. The general format is **`schedule(kind of schedule, chunk size)`**

• **`static`**

Chunks of iterations are assigned to the threads in cyclic order. Size of default chunk, roughly = n / number of threads.

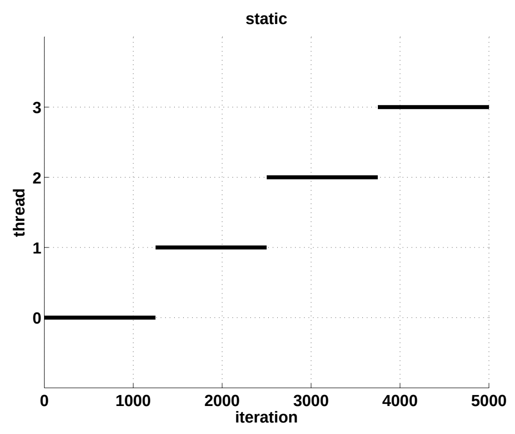
Low overhead, good if the same amount of work in each iteration. **`chunk`** can be used to access array elements in groups (may be me more efficient, e.g. using cache memories in better way).

Here is a small example:

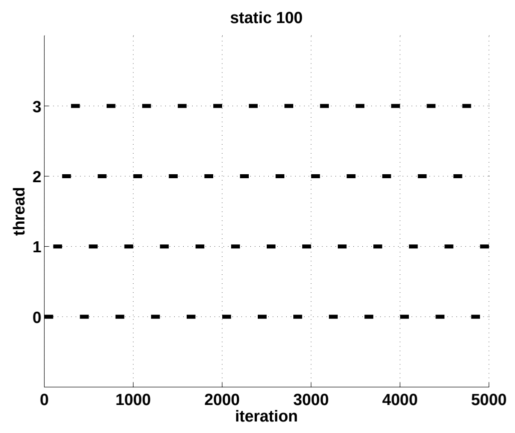
```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(static, 4)  ! 4 = chunk
  do k = 1, n
    ...
  end do
```

```
                            1                   2
k        : 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
thread 0: x x x x                 x x x x
thread 1:         x x x x                 x x x x
thread 2:                 x x x x
```

Here is a larger problem, where $n = 5000$, **`schedule(static)`**, and using four threads.



`schedule(static, 100)`

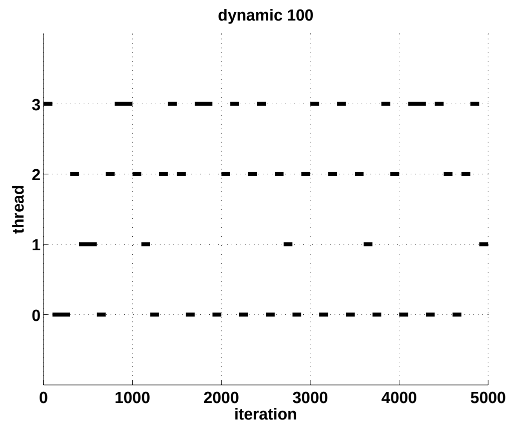


Note that if the chunk size is 5000 (in this example) only the first thread would work, so the chunk size should be chosen relative to the number of iterations.

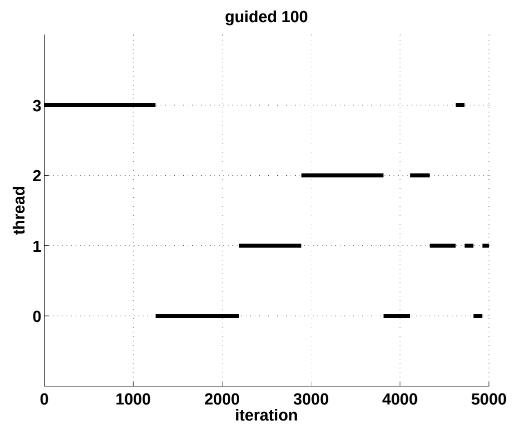
• **`dynamic`**
If the amount of work varies between iterations we should use **`dynamic`** or **`guided`**. With **`dynamic`**, threads compete for **`chunk`**-sized assignments. Note that there is a synchronization overhead for **`dynamic`** and **`guided`**.

```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(dynamic, chunk)
 ...
```

Here a run with **`schedule(dynamic,100)`**(**`schedule(dynamic)`**, gives a chunks size of one). The amount of works differs between iterations in the following examples.



• **`guided`**
There is also **`schedule(guided, chunk)`** assigning pieces of work (≥ **`chunk`**) proportional to the number of remaining iterations divided by the number of threads.
Large chunks in the beginning smaller at the end. It requires fewer synchronisations than **`dynamic`**.



• **`runtime`**
It is also possible to decide the scheduling at runtime, using an environment variable, **`OMP_SCHEDULE`**, e.g.

```
!$omp parallel do private(k) shared(x, n) &
!$omp          schedule(runtime)

...

  setenv OMP_SCHEDULE "guided,100"    tcsh
  export OMP_SCHEDULE=dynamic         bash
```

Suppose we parallelise m iterations over P processors.
No default scheduling is defined in the OpenMP-standard, but **schedule(static, m / P)**is a common choice (assuming that P divides m).

Here comes an example where this strategy works badly.
So do not always use the standard choice.

We have nested loops, where the number of iterations in the inner loop depends on the loop index in the outer loop.

```
!$omp ...
do j = 1, m           ! parallelise this loop
  do k = j + 1, m     ! NOTE: k = j + 1
     call work(...)   ! each call takes the same time
  end do
end do
```

Suppose m is large and let T_{ser} be the total run time on one thread. If there is no overhead, the time, T_t, for thread number t is approximately:

$$T_t \approx \frac{2T_{ser}}{P}\left(1 - \frac{t + 1/2}{P}\right), \quad t = 0, \ldots, P-1$$

So thread zero has much more work to do compared to the last thread:

$$\frac{T_0}{T_{P-1}} \approx 2P - 1$$

a very poor balance. The speedup is bounded by T_0:

$$\text{speedup} = \frac{T_{ser}}{T_0} \approx \frac{P}{2 - 1/P} \approx \frac{P}{2}$$

and not the optimal P.

Here is a test:

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

void work(double *);

int main()
{
  const int M = 1000, MAX_THREADS = 4;
  double s[MAX_THREADS - 1], time;
  int j, k, i_am, thr;

  for (thr = 1; thr <= MAX_THREADS; thr++) {
    omp_set_num_threads(thr);

    time = omp_get_wtime();  // a builtin function
    #pragma omp parallel private(j, k, i_am) shared(s)
    {
      i_am = omp_get_thread_num();

      #pragma omp for schedule(runtime)
      for (j = 1; j <= M; j++)
        for (k = j + 1; k <= M; k++)
          work(&s[i_am]);
    }

    printf("time = %4.2f\n", omp_get_wtime() - time);
  }

  for (j = 0; j < MAX_THREADS; j++)
    printf("%e ", s[j]);
  printf("\n");

  return 0;
}
```

```
void work(double *s)
{
  int k;

  *s = 0.0;
  for (k = 1; k <= 1000; k++)
    *s += 1.0 / k;
}
% icc -O3 -openmp load_bal.c
% setenv OMP_SCHEDULE static
% a.out      run on Ferlin (edited)
time = 3.83
time = 2.88
time = 2.13
time = 1.68
time = 1.38
time = 1.17
time = 1.02
time = 0.90

% setenv OMP_SCHEDULE "static,10"
time = 3.83
time = 1.94
time = 1.30
time = 0.99
time = 0.80
time = 0.67
time = 0.58
time = 0.51
```

dynamic and **guided** give the same times as **static,10**, in this case. A chunk size of 1-20 works well, but more than 50 gives longer execution times.

Note that $P/(2 - 1/P) \approx 4.3$ and $3.83/0.9 \approx 4.26$ and $3.83/0.51 \approx 7.5$. So the analysis is quite accurate in this simple case.

Do not misuse **dynamic**. Here is a contrived example:

```
...

  int k, i_am, iter[] = { 0, 0, 0, 0 };
  double time;

  omp_set_num_threads(4);
  time = omp_get_wtime();

  #pragma omp parallel private(k, i_am) shared(iter)
  {
    i_am = omp_get_thread_num();

    #pragma omp for schedule(runtime)
    for (k = 1; k <= 100000000; k++)
      iter[i_am]++;
  }
  printf("time: %5.2f, iter: %d %d %d %d\n",
         omp_get_wtime() - time,
         iter[0], iter[1], iter[2], iter[3]);
...
ferlin > setenv OMP_SCHEDULE static
time:  0.01, iter: 25000000 25000000 25000000 25000000

ferlin > setenv OMP_SCHEDULE dynamic
time: 15.53, iter: 25611510 25229796 25207715 23950979

ferlin > setenv OMP_SCHEDULE "dynamic,10"
time:  1.32, iter: 25509310 24892310 25799640 23798740

ferlin > setenv OMP_SCHEDULE "dynamic,100"
time:  0.13, iter: 29569500 24044300 23285700 23100500

ferlin > setenv OMP_SCHEDULE guided
time:  0.00, iter = 39831740 5928451 19761833 34477976
```

The reduction clause

```
...
  int i, n = 10000;
  double x[n], y[n], s;

  for (i = 0; i < n; i++) {
    x[i] = 1.0; y[i] = 2.0;   // Init.
  }
   s = 0.0;
  #pragma omp parallel for private(i) shared(n, x, y)
  reduction(+: s)   // all on the same line
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
...
```

In general: **reduction(operator: variable list)**
Valid operators are: **+, *, -, &, |, ^, &&, ||**.
A reduction is typically specified for statements of the form:

```
  x = x op expr
  x = expr op x  (except for subtraction)
  x binop= expr
  x++
  ++x
  x--
  --x
```

where **expr** is of scalar type and does not reference **x**.
This is what happens in our example above:

- each thread gets its local sum-variable, $s_{\#thread}$ say
- $s_{\#thread} = 0$ before the loop (the thread private variables are initialised in different ways depending on the operation, zero for **+** and **-**, one for *****). See the standard for the other cases.
- each thread computes its sum in $s_{\#thread}$
- after the loop all the $s_{\#thread}$ are added to **s** in a safe way

In Fortran:
reduction(operator or intrinsic: variable list)

Valid operators are: **+, *, -, .and., .or., .eqv., .neqv.** and intrinsics: **max, min, iand, ior, ieor**(the **iand** is bit-wise and, etc.)

The operator/intrinsic is used in one of the following ways:

- **x = x operator expression**
- **x = expression operator x**(except for subtraction)
- **x = intrinsic(x, expression)**
- **x = intrinsic(expression, x)**

where **expression** does not involve **x**.

Note that **x** may be an array in Fortran (vector reduction) but not so in C. Here is a contrived example which computes a matrix-vector product:

```
...
  double precision, dimension(n, n) :: A
  double precision, dimension(n)    :: s, x

  A = ...  ! initialize A and b
  b = ...
  s = 0.0d0

  !$omp parallel do shared(A, x) reduction(+: s) &
  !$omp  private(i) default(none)
  do i = 1, n
    s = s + A(:, i) * x(i)  ! s = A * x
  end do
...
```

We can implement our summation example without using **reduction**-variables. The problem is to update the shared sum in a safe way. This can be done using critical sections.

```
...
  double private_s, shared_s;
...

  shared_s = 0.0;

// a parallel region
#pragma omp parallel private(private_s)
            shared(x, y, shared_s, n)
  {
    private_s = 0.0;  // Done by each thread

    #pragma omp for private(i)  // A parallel loop
    for (i = 0; i < n; i++)
      private_s += x[i] * y[i];

    // Here we specify a critical section.
    // Only one thread at a time may pass through.

    #pragma omp critical
    shared_s += private_s;
  }
...
```

Vector reduction in C

Here are two alternatives in C (and Fortran if the compiler does not support vector reduction).

We introduce a private summation vector, **partial_sum**, one for each thread.

```
...
  for(k = 0; k < n; k++)  // done by the master
    shared_sum[k] = 0.0;

  #pragma omp parallel private(partial_sum, k) \
                     shared(shared_sum) ...
  {
  // Each thread updates its own partial_sum.
  // We assume that this is the time consuming part.
    for( ...
      partial_sum[k] = ..
  ...

  // Update the shared sum in a safe way.
  // Not too bad with a critical section here.
    #pragma omp critical
    {
      for(k = 0; k < n; k++)
        shared_sum[k] += partial_sum[k];
    }
  } // end parallel
...
```

We can avoid the critical section if we introduce a shared matrix where each row (or column) corresponds to the **partial_sum** from the previous example.

```
...
  for(k = 0; k < n; k++)  // done by the master
    shared_sum[k] = 0.0;

  #pragma omp parallel private(i_am) \
              shared(S, shared_sum, n_threads)
  {
    i_am = omp_get_thread_num();

    for(k = 0; k < n; k++)  // done by all
      S[i_am][k] = 0.0;

   // Each thread updates its own partial_sum.
   // We assume that this is the time consuming part.
    for( ...
      S[i_am][k] = ..

   // Add the partial sums together.
   // The final sum could be stored in S of course.
    #pragma omp for
    for(k = 0; k < n; k++)
      for(j = 0; j < n_threads; j++)
        shared_sum[k] += S[j][k];

  } // end parallel
...
```

Nested loops, matrix-vector multiply

```
a = 0.0
do j = 1, n
  do i = 1, m
    a(i) = a(i) + C(i, j) * b(j)
  end  do
end do
```

Can be parallelised with respect to **i** but not with respect to **j** (since different threads will write to the same **a(i)**).

May be inefficient since parallel execution is initiated **n** times (procedure calls). OK if **n** small and **m** large.

Switch loops.

```
a = 0.0
do i = 1, m
  do j = 1, n
    a(i) = a(i) + C(i, j) * b(j)
  end  do
end do
```

The **do i** can be parallelised. Bad cache locality for **C**.

Test on Ferlin using **ifort -O3 ...**. The loops were run ten times. Times in seconds for one to four threads. **dgemv** from MKL takes 0.23s, 0.21s, 0.34s for the three cases and the builtin **matmul** takes 1.0s, 0.74s, 1.1s. *Use BLAS!*

m	n	first loop				second loop			
		1	2	3	4	1	2	3	4
4000	4000	0.41	0.37	0.36	0.36	2.1	1.2	0.98	0.83
40000	400	0.39	0.32	0.27	0.23	1.5	0.86	0.72	0.58
400	40000	0.49	1.2	1.5	1.7	1.9	2.0	2.3	2.3

- Cache locality is important.
- If second loop is necessary, OpenMP gives speedup.
- Large n gives slowdown in first loop.

A few other OpenMP directives, C

```
#pragma omp parallel shared(a, n)

        ... code run in parallel

#pragma omp single   // only ONE thread will
{                    // execute the code
        ... code
}

#pragma omp barrier  // wait for all the other threads
      ... code

// don’t wait (to wait is default)
#pragma omp for nowait
    for ( ...

    for ( ... // all iterations run by all threads

#pragma omp sections
{
#pragma omp section
          ... code executed by one thread
#pragma omp section
          ... code executed by another thread
} // end sections, implicit barrier

# ifdef _OPENMP
  C statements ...  Included if we use OpenMP,
  but not otherwise (conditional compilation)
# endif

} // end of the parallel section
```

A few other OpenMP directives, Fortran

```
!$omp parallel shared(a, n)    ! a parallel region

        ... code run in parallel

!$omp single   ! only ONE thread will execute the code
        ... code
!$omp end single

!$omp barrier  ! wait for all the other threads
      ... code

!$omp do private(k)
        do ...
        end do
!$omp end do nowait  ! don’t wait (to wait is default)

      do ... ! all iterations run by all threads
      end do

!$omp sections
!$omp   section
          ... code executed by one thread
!$omp   section
          ... code executed by another thread
!$omp end sections  ! implicit barrier

!$ Fortran statements ...  Included if we use OpenMP,
!$ but not otherwise (conditional compilation)

!$omp end parallel  ! end of the parallel section
```

Misuse of critical, atomic

Do not use critical sections and similar constructions too much. This test compares three ways to compute a sum. We try reduction, critical and atomic. $n = 10^7$.

```
...
  printf("n_thr    time, reduction\n");
  for(n_thr = 1; n_thr <= 4; n_thr++) {
    omp_set_num_threads(n_thr);
    s = 0.0;
    t = omp_get_wtime();

    #pragma omp parallel for reduction(+: s) private(i)
    for (i = 1; i <= n; i++)
      s += sqrt(i);

    printf("%3d %10.3f\n", n_thr, omp_get_wtime() - t);
  }
  printf("s = %e\n", s);
```

Change the inner loop to

```
    #pragma omp parallel for shared(s) private(i)
    for (i = 1; i <= n; i++) {
      #pragma omp critical
      s += sqrt(i);
    }
```

and then to

```
    #pragma omp parallel for shared(s) private(i)
    for (i = 1; i <= n; i++) {
      #pragma omp atomic
      s += sqrt(i);
    }
```

`atomic` updates a single variable atomically.

Here are the times (on Ferlin):

```
n_thr    time, reduction
  1      0.036
  2      0.020
  3      0.014
  4      0.010

n_thr    time, critical
  1      0.666
  2      5.565
  3      5.558
  4      5.296

n_thr    time, atomic
  1      0.188
  2      0.537
  3      0.842
  4      1.141
```

We get a slowdown instead of a speedup, when using `critical` or `atomic`.

workshare

Some, but not all, compilers support parallelisation of Fortran90 array operations, e.g.

```
  ... code
! a, b and c are arrays

!$omp parallel shared(a, b, c)
!$omp   workshare
          a = 2.0 * cos(b) + 3.0 * sin(c)
!$omp   end workshare
!$omp end parallel
  ... code
```

or shorter

```
  ... code
!$omp parallel workshare shared(a, b, c)
        a = 2.0 * cos(b) + 3.0 * sin(c)
!$omp end parallel workshare
  ... code
```

Subroutines and OpenMP

Here comes a first example of where we call a subroutine from a parallel region. If we have time leftover there will be more at the end of the lecture. Formal arguments of called routines, that are passed by reference, inherit the data-sharing attributes of the associated actual parameters. Those that are passed by value become private. So:

```
  void work( double [], double [], double, double,
             double *,  double *);
  ...
  double  pr_vec[10], sh_vec[10], pr_val, sh_val,
          pr_ref,     sh_ref;
  ...

#pragma omp parallel private(pr_vec, pr_val, pr_ref)
                     shared( sh_vec, sh_val, sh_ref)
{
  work(pr_vec, sh_vec, pr_val, sh_val, &pr_ref, &sh_ref
}
...

void work(double pr_vec[], double sh_vec[],
          double pr_val,   double sh_val,
          double *pr_ref,  double *sh_ref)
{
  // pr_vec becomes private
  // sh_vec becomes shared
  // pr_val becomes private
  // sh_val becomes PRIVATE, each thread has its own
  // pr_ref becomes private
  // sh_ref becomes shared

  int k;  // becomes private
 ...
}
```

In Fortran alla variables are passed by reference, so they inherit the data-sharing attributes of the associated actual parameters.

Here comes a simple example in C:

```
#include <stdio.h>
#include <omp.h>
void work(int[], int);

int main()
{
  int a[] = { 99, 99, 99, 99 }, i_am;

  omp_set_num_threads(4);

  #pragma omp parallel private(i_am) shared(a)
  {
    i_am = omp_get_thread_num();
    work(a, i_am);

    #pragma omp single
     printf("a = %d, %d, %d, %d\n",
             a[0], a[1], a[2], a[3]);
  }
  return 0;
}

// a[] becomes shared, i_am becomes private

void work(int a[], int i_am)
{
  int k; // becomes private (not used in this example)

  printf("work %d\n", i_am);
  a[i_am] = i_am;
}
```

```
% a.out
 work 1
 work 3
 a =  99,  1,  99,  3
 work 2
 work 0
```

Print after the parallel region or add a barrier:

```
#pragma omp barrier
#pragma omp single
    printf("a = %d, %d, %d, %d\n",
            a[0], a[1], a[2], a[3]);
% a.out
 work 0
 work 1
 work 3
 work 2
 a =  0,  1,  2,  3
```

OpenMP makes no guarantee that input or output to the same file is synchronous when executed in parallel. You may need to link with a special thread safe I/O-library.

Matlab and parallel computing

Two major options.

1. Message passing by using the "Distributed Computing Toolbox" (a large toolbox, the User's Guide is 529 pages).
2. Threads & shared memory by using the parallel capabilities of the underlying numerical libraries.

We are only going to look at threads.
Threads can be switched on in two ways. From the GUI: Preferences/General/Multithreading or by using **maxNumCompThreads** Here is a small example:

```
T = [];
for thr = 1:4
  maxNumCompThreads(thr);  % set #threads
  j = 1;
  for n = [800 1600 3200]
    A = randn(n);
    B = randn(n);
    t = clock;
      C = A * B;
    T(thr, j) = etime(clock, t);
    j = j + 1;
  end
end
```

The speedup depends on the library. This is how you can find out which library is used:

```
% setenv LAPACK_VERBOSITY 1
% matlab
cpu_id: x86 Family 15 Model 1 Stepping 0, AuthenticAMD
  etc.
libmwblas: loading acml.so
libmwblas: resolved caxpy_ in 0x524c10
etc.
```

So ACML (AMD Core Math Library) is used in this case.
Another alternative is to use MKL
(see the gui-help for **BLAS_VERSION**).

We tested solving linear systems and computing eigenvalues as well. Here are the times using one to four threads:

n	C = A * B				x = A \ b				l = eig(A)			
	1	2	3	4	1	2	3	4	1	2	3	4
800	0.3	0.2	0.1	0.1	0.2	0.1	0.1	0.1	3.3	2.5	2.4	2.3
1600	2.1	1.1	0.8	0.6	1.1	0.7	0.6	0.5	20	12	12	12
3200	17.0	8.5	6.0	4.6	7.9	4.8	4.0	3.5	120	87	81	80

So, using several threads can be an option if we have a large problem. We get a better speedup for the multiplication, than for **eig**, which seems reasonable.

This method can be used to speed up the computation of elementary functions as well.

A parallel library

Libraries like ACML and MKL can be used from C and Fortran as well. I am using an Opteron and fetched AMD's ACML-library. **dgesv** is a Lapack-routine for solving a general linear system.

```
% cat dgesv_ex.f90
  ...
! A is a 2000 x 2000-matrix

  t = fsecond()  ! my own routine
    call dgesv ( n, nrhs, A, lda, ipiv, b, ldb, info )
  t = fsecond() - t

  print '(a, f5.2)', 'time = ', t
  ...

% gfortran -O3 dgesv_ex.f90 fsecond.o  \
   -Lgfortran64_mp/lib -lacml_mp -lacml_mv
% setenv LD_LIBRARY_PATH ...

% setenv OMP_NUM_THREADS 1
% a.out
time =  1.56
time =  0.84   on 2 threads
time =  0.58   on 3 threads
time =  0.46   on 4 threads
```

Increasing n to 4000 we get the times:

```
time = 11.92
time =  6.09
time =  4.18
time =  3.23
```

Case study: solving a large and stiff IVP

$$y'(t) = f(t, y(t)),\ y(0) = y_0,\ y,\ y_0 \in \Re^n,\quad f : \Re \times \Re^n \to \Re^n$$

where $f(t, y)$ is expensive to evaluate.

LSODE (Livermore Solver for ODE, Alan Hindmarsh) from netlib. BDF routines; Backward Differentiation Formulas.

Implicit method: t_k present time, $y^{(k)}$ approximation of $y(t_k)$.

Backward Euler (simplest BDF-method). Find $y^{(k+1)}$ such that:

$$y^{(k+1)} = y^{(k)} + hf(t_{k+1}, y^{(k+1)})$$

LSODE is adaptive (can change both h and the order).

Use Newton's method to solve for $z \equiv y^{(k+1)}$:

$$z - y^{(k)} - hf(t_{k+1}, z) = 0$$

One step of Newton's method reads:

$$z^{(i+1)} = z^{(i)} - \left[I - h\frac{\partial f}{\partial y}(t_{k+1}, z^{(i)})\right]^{-1} (z^{(i)} - y^{(k)} - hf(t_{k+1}, z^{(i)}))$$

The Jacobian $\frac{\partial f}{\partial y}$ is approximated by finite differences one column at a time. Each Jacobian requires n evaluations of f.

$$\frac{\partial f}{\partial y}\, e_j \approx \left[f(t_{k+1}, z^{(i)} + e_j\delta_j) - f(t_{k+1}, z^{(i)})\right] / \delta_j$$

e_j is column j in the identity matrix I.

Parallelise the computation of the Jacobian, by computing columns in parallel. Embarrassingly parallel.

Major costs in LSODE:

1. Computing the Jacobian, J, (provided f takes time).
2. LU-factorization of the Jacobian (once for each time step).
3. Solving the linear systems, given L and U.

What speedup can we expect?

Disregarding communication, the wall clock time for p threads, looks something like (if we compute J in parallel):

$$wct(p) = time(\text{LU}) + time(\text{solve}) + \frac{time(\text{computing J})}{p}$$

If the parallel part, "computing J", dominates we expect good speedup at least for small p. Speedup may be close to linear, $wct(p) = wct(1)/p$.

For large p the serial (non-parallel) part will start to dominate.

How should we speed up the serial part?

1. Switch from Linpack, used in LSODE, to Lapack.
2. Try to use a parallel library like ACML.

After having searched LSODE (Fortran 66):

```
c if miter = 2, make n calls to f to approximate j.
 ...
     j1 = 2
     do 230 j = 1,n
       yj = y(j)
       r = dmax1(srur*dabs(yj),r0/ewt(j))
       y(j) = y(j) + r
       fac = -hl0/r
       call f (neq, tn, y, ftem)
       do 220 i = 1,n
220      wm(i+j1) = (ftem(i) - savf(i))*fac
       y(j) = yj
       j1 = j1 + n
230    continue
 ...
c add identity matrix.
 ...
c do lu decomposition on p.
      call dgefa (wm(3), n, n, iwm(21), ier)
...
 100  call dgesl (wm(3), n, n, iwm(21), x, 0)
```

We see that
r $= \delta_j$
fac $= -h/\delta_j$
tn $= t_{k+1}$
ftem $= f(t_{k+1}, z^{(i)} + e_j\delta_j)$
wm(2...) is the approximation to the Jacobian.

From reading the code: **neq** is an array but **neq(1) = n**

The parallel version

- `j, i, yj, r, fac, ftem` are private
  `ftem` is the output (y') from the subroutine
- `j1 = 2` offset in the Jacobian; use `wm(i+2+(j-1)*n)`
  no index conflicts
- `srur, r0, ewt, hl0, wm, savf, n, tn` are shared
- `y` is a problem since it is modified. `shared` does not work. `private(y)` will not work either; we get an uninitialised copy. `firstprivate` is the proper choice, it makes a private and initialised copy.

```
c$omp  parallel do private(j, yj, r, fac, ftem)
c$omp+ shared(f, srur, r0, ewt, hl0, wm, savf,n,neq,tn)
c$omp+ firstprivate(y)
      do j = 1,n
        yj = y(j)
        r = dmax1(srur*dabs(yj),r0/ewt(j))
        y(j) = y(j) + r
        fac = -hl0/r
        call f (neq, tn, y, ftem)
        do i = 1,n
          wm(i+2+(j-1)*n) = (ftem(i) - savf(i))*fac
        end do
        y(j) = yj
      end do
```

Did not converge! After reading of the code:

```
 dimension neq(1), y(1), yh(nyh,1), ewt(1), ftem(1)
change to
 dimension neq(1), y(n), yh(nyh,1), ewt(1), ftem(n)
```

More on OpenMP and subprograms

So far we have essentially executed a main program containing OpenMP-directives. Suppose now that we call a function, containing OpenMP-directives, from a parallel part of the program, so something like:

```
int main()
{
  ...
  #pragma omp parallel ...  ---
  {                           |
    #pragma omp for ...       | lexical extent of
    ...                       |
    work(...);                | the parallel region
    ...                       |
  }                         ---
  ...
}

void work(...)
{
  ...
  #pragma omp for            ---
  for (...) {                  | dynamic extent of the
     ...                       | parallel region
    }                        ---
  ...
}
```

The `omp for` in `work` is an orphaned directive (it appears in the dynamic extent of the parallel region but not in the lexical extent). This `for` binds to the dynamically enclosing parallel directive and so the iterations in the `for` will be done in parallel (they will be divided between threads).

Suppose now that `work` contains the following three loops and that we have three threads:

```
...
  int k, i_am;
  char f[] = "%1d:%5d %5d %5d\n";  // a format

  #pragma omp master
  printf("     i_am   omp()  k\n");

  i_am = omp_get_thread_num();

  #pragma omp for private(k)
  for (k = 1; k <= 6; k++)      // LOOP 1
    printf(f, 1, i_am, omp_get_thread_num(), k);

  for (k = 1; k <= 6; k++)      // LOOP 2
    printf(f, 2, i_am, omp_get_thread_num(), k);

  #pragma omp parallel for private(k)
  for (k = 1; k <= 6; k++)      // LOOP 3
    printf(f, 3, i_am, omp_get_thread_num(), k);
...
```

In `LOOP 1` thread 0 will do the first two iterations, thread 1 performs the following two and thread 2 takes the last two.
In `LOOP 2` all threads will do the full six iterations.
In the third case we have:

> A `PARALLEL` directive dynamically inside another `PARALLEL` directive logically establishes a new team, which is composed of only the current thread, unless nested parallelism is established.

We say that the loops is serialised. All threads perform six iterations each.

If we want the iterations to be shared between new threads we can set an environment variable, `setenv OMP_NESTED TRUE`, or `omp_set_nested(1)`.
If we enable nested parallelism we get three teams consisting of three threads each, in this example.

This is what the (edited) printout from the different loops may look like. `omp()` is the value returned by `omp_get_thread_num()`.
The output from the loops may be interlaced though.

```
     i_am  omp() k            i_am  omp() k
  1:    1    1    3        3:    1    0    1
  1:    1    1    4        3:    1    0    2
  1:    2    2    5        3:    1    2    5
  1:    2    2    6        3:    1    2    6
  1:    0    0    1        3:    1    1    3
  1:    0    0    2        3:    1    1    4
                           3:    2    0    1
  2:    0    0    1        3:    2    0    2
  2:    1    1    1        3:    2    1    3
  2:    1    1    2        3:    2    1    4
  2:    2    2    1        3:    2    2    5
  2:    0    0    2        3:    2    2    6
  2:    0    0    3        3:    0    0    1
  2:    1    1    3        3:    0    0    2
  2:    1    1    4        3:    0    1    3
  2:    1    1    5        3:    0    1    4
  2:    1    1    6        3:    0    2    5
  2:    2    2    2        3:    0    2    6
  2:    2    2    3
  2:    2    2    4
  2:    2    2    5
  2:    2    2    6
  2:    0    0    4
  2:    0    0    5
  2:    0    0    6
```

Case study: sparse matrix multiplication

Task: given a matrix A which is large, sparse and symmetric we want to:

- compute a few of its smallest eigenvalues OR
- solve the linear system $Ax = b$

n is the dimension of A and nz is the number of nonzeros.

Some background, which you may read after the lecture:

We will study iterative algorithms based on forming the Krylov subspace: $\{v, Av, A^2v, \dots, A^{j-1}v\}$. v is a random-vector. So, Paige-style Lanczos for the eigenvalue problem and the conjugate-gradient method for the linear system, for example. When solving $Ax = b$ we probably have a preconditioner as well, but let us skip that part.

The vectors in the Krylov subspace tend to become almost linearly dependent so we compute an orthonormal basis of the subspace using Gram-Schmidt. Store the basis-vectors as columns in the $n \times j$-matrix V_j.

Project the problem onto the subspace, forming $T_j = V_j^T A V_j$ (tridiagonal) and solve the appropriate smaller problem, then transform back.

T_j and the basis-vectors can be formed as we iterate on j. In exact arithmetic it is sufficient to store the three latest v-vectors in each iteration.

p is the maximum number of iterations.

A Lanczos-algorithm may look something like:

```
                                                  # operations
v = randn(n, 1)                                   $\mathcal{O}(n)$
$v = v/||v||_2$                                   $\mathcal{O}(n)$
for $j = 1$ to $p$ do
  $t = Av$                                        $\mathcal{O}(nz)$
  if $j > 1$ then $t = t - \beta_{j-1} w$ endif   $\mathcal{O}(n)$
  $\alpha_j = t^T v$                              $\mathcal{O}(n)$
  $t = t - \alpha_j v$                            $\mathcal{O}(n)$
  $\beta_j = ||t||_2$                             $\mathcal{O}(n)$
  $w = v$                                         $\mathcal{O}(n)$
  $v = t/\beta_j$                                 $\mathcal{O}(n)$
  Solve the projected problem and                 $\mathcal{O}(j)$
  and check for convergence
end for
```

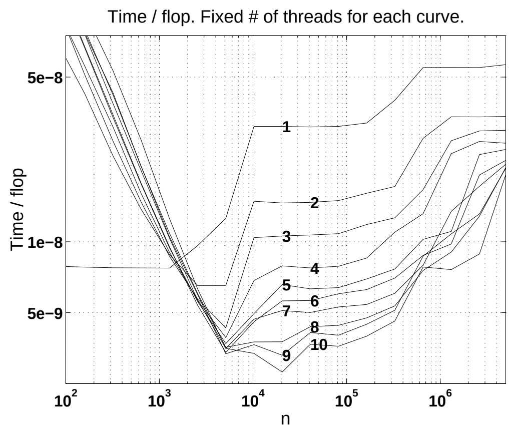
The diagonal of T_j is $\alpha_1, \dots, \alpha_j$ and the sub- and super-diagonals contain $\beta_1, \dots, \beta_{j-1}$.

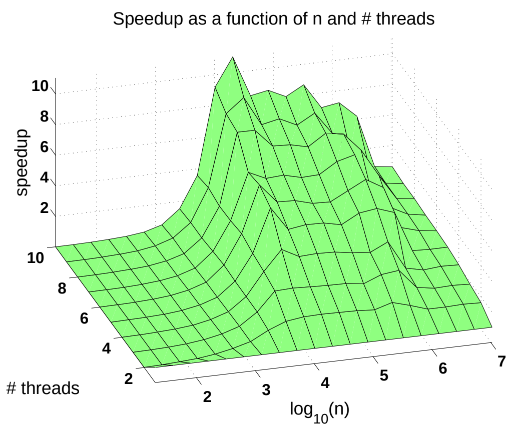
How can we parallelise this algorithm?

- The j-iterations and the statements in each iteration must be done in order. Not possible to parallelise.
- It is easy to parallelise each of the simple vector operations (the ones that cost $\mathcal{O}(n)$). May not give any speedup though.
- The expensive operation in an iteration is usually Av.
- Solving the projected problem is rather fast and not so easy to parallelise (let us forget it).

We will not look at graph-based pre-ordering algorithms.
A block diagonal matrix would be convenient, for example.

Vectors must not be too short if we are going to succeed.
The figures show how boye (SGI) computes **daxpy** for different n and number of threads.





The tricky part, parallelising $t = Av$

A is large, sparse and symmetric so we need a special data structure which takes the sparsity and the symmetry into account.

First try: store all triples $(r, c, a_{r,c})$ where $a_{r,c} \neq 0$ and $r \leq c$. I.e. we are storing the nonzeros in the upper triangle of the matrix.

The triples can be stored in three arrays, **rows**, **cols** and **A** or as an array of triples. Let us use the three arrays and let us change the meaning of **nz** to mean the number of stored nonzeros. The first coding attempt may look like:

```
do k = 1, nz
  if ( rows(k) == cols(k) ) then
    ...                  ! diagonal element
  else
    ...                  ! off-diagonal element
  end if
end do
```

If-statements in loops may degrade performance, so we must think some more.

If A has a dense diagonal we can store it in a separate array, **diag_A** say. We use the triples for all $a_{r,c} \neq 0$ and $r < c$ (i.e. elements in the strictly upper triangle).

If the diagonal is sparse we can use pairs $(r, a_{r,r})$ where $a_{r,r} \neq 0$. Another way is to use the triples format but store the diagonal first, or to store $a_{k,k}/2$ instead of $a_{k,k}$.

Our second try may look like this, where now `nz` is the number stored nonzeros in the strictly upper triangle of A.

```
! compute t = diag(A) * v
 ...

  do k = 1, nz  ! take care of the off-diagonals
    r    = rows(k)
    c    = cols(k)
    t(r) = t(r) + A(k) * v(c)  ! upper triangle
    t(c) = t(c) + A(k) * v(r)  ! lower triangle
  end do
```

$$\begin{bmatrix} \vdots \\ t_r \\ \vdots \\ t_c \\ \vdots \end{bmatrix} = \begin{bmatrix} \ddots & \vdots & & \vdots & \\ \dots & a_{r,r} & \dots & a_{r,c} & \dots \\ & \vdots & \ddots & \vdots & \\ \dots & a_{c,r} & \dots & a_{c,c} & \dots \\ & \vdots & & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ v_r \\ \vdots \\ v_c \\ \vdots \end{bmatrix}$$

Let us now concentrate on the loops for the off-diagonals and make it parallel using OpenMP.

Note that we access the elements in A once.

```
! Take care of diag(A)
 ...
  !$omp do default(none), private(k, r, c), &
  !$omp    shared(rows, cols, A, nz, v, t)
  do k = 1, nz  ! take care of the off-diagonals
    r    = rows(k)
    c    = cols(k)
    t(r) = t(r) + A(k) * v(c)  ! upper triangle
    t(c) = t(c) + A(k) * v(r)  ! lower triangle
  end do
```

This will probably give us the wrong answer (if we use more than one thread) since two threads can try to update the same `t`-element.

Example: The first row in A it will affect t_1, t_3 and t_5, and the second row in A will affect t_2, t_4 and t_5. So there is a potential conflict when updating t_5 if the two rows are handled by different threads.

$$\begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & a_{1,3} & 0 & a_{1,5} \\ 0 & 0 & 0 & a_{2,4} & a_{2,5} \\ a_{1,3} & 0 & 0 & 0 & 0 \\ 0 & a_{2,4} & 0 & 0 & 0 \\ a_{1,5} & a_{2,5} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}$$

If the first row is full it will affect all the other rows. A block diagonal matrix would be nice.

As in the previous example it is not possible to use critical sections. Vector reduction is an option and we can do our own as before. Here is a version using a public matrix.

`X` has n rows and as many columns as there are threads, `num_thr` below. Each thread stores its sum in `X(:, thr)`, where `thr` is the index of a particular thread.

Here is the code:

```
!$omp parallel shared(X, ...)
      ...
      i_am = omp_get_thread_num() + 1
      ...
      do i = 1, n        ! done by all threads
        X(i, i_am) = 0.0 ! one column each
      end do

      !$omp do
        do i = 1, nz
          r = rows(i)
          c = cols(i)
          X(r, i_am) = X(r, i_am) + A(i)* v(c)
          X(c, i_am) = X(c, i_am) + A(i)* v(r)
        end do
      !$omp end do

      !$omp do
       do i = 1, n
          do thr = 1, num_thr
            t(i) = t(i) + X(i, thr)
          end do
        end do
      ...
!$omp end parallel
```

The addition loop is now parallel, but we have bad cache locality when accessing `X` (this can be fixed). None of the parallel loops should end with `nowait`.
One can get a reasonable speedup (depends on problem and system).

Compressed storage

The triples-format is not the most compact possible. A common format is the following compressed form. We store the diagonal separately as before and the off-diagonals are stored in order, one row after the other. We store `cols` as before, but `rows` now points into `cols` and `A` where each new row begins. Here is an example (only the strictly upper triangle is shown):

$$\begin{bmatrix} 0 & a_{1,2} & a_{1,3} & 0 & a_{1,5} \\ 0 & 0 & a_{2,3} & a_{2,4} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{4,5} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

is stored as `A` $= [\ a_{1,2}\ a_{1,3}\ a_{1,5} \mid a_{2,3}\ a_{2,4} \mid 0 \mid a_{4,5}\ ]$,
`cols` $= [\ 2\ 3\ 5 \mid 3\ 4 \mid \bullet \mid 5\ ]$, ($\bullet$ fairly arbitrary, n say)
`rows` $= [\ 1\ 4\ 6\ 7\ 8\ ]$. (8 is one step after the last)

Note that `rows` now only contains n elements.
The multiplication can be coded like this (no OpenMP yet):

```
   ... take care of diagonal, t = diag(A)* v

  do r = 1, n - 1  ! take care of the off-diagonals
    do k = rows(r), rows(r + 1) - 1
      c    = cols(k)
      t(r) = t(r) + A(k) * v(c)  ! upper triangle
      t(c) = t(c) + A(k) * v(r)  ! lower triangle
    end do
  end do
```

We can parallelise this loop (with respect to `do r`) in the same way as we handled the previous one (using the extra array `X`).

There is one additional problem though.
Suppose that the number of nonzeros per row is fairly constant and that the nonzeros in a row is evenly distributed over the columns.

If we use default static scheduling the iterations are divided among the threads in contiguous pieces, and one piece is assigned to each thread. This will lead to a load imbalance, since the upper triangle becomes narrower for increasing `r`.

To make this effect very clear I am using a full matrix (stored using a sparse format).

A hundred matrix-vector multiplies with a full matrix of order 2000 takes (wall-clock-times):

#threads →	1	2	3	4
triple storage	19.7	10.1	7.1	6.9
compressed, static	20.1	16.6	12.6	10.1
compressed, static, 10	20.1	11.2	8.8	7.5

The time when using no OpenMP is essentially equal to the time for one thread.

Conclusions

- Optimise for one processor first. If the code still is too slow, parallelise it. A parallel code can be much faster.
- Profile your code to find the computationally expensive parts that can be run in parallel.
- Use the optimisation flags and OpenMP-directives.
- Do you get reasonable speedup, reasonable Gflop?
- Do you get the correct results (for different number of threads)?
- Use the tuned numerical libraries, perhaps there are parallel routines.
- When you parallelise:
  - try do avoid synchronisation (`barrier`, `critical`, `atomic`)
  - try do avoid `single`-sections
  - examine the different ways the loop can be parallelised
  - do not forget single-CPU optimisation (cache locality)
  - choose a suitable schedule