OpenMP - shared memory parallelism

Thomas Ericsson

Computational Mathematics
Chalmers University of Technology
and the University of Gothenburg

PDC Summer School 2013

Table of Contents I

Table of Contents II

Table of Contents III

5: Contents

- Some important concepts in parallel computing.
- OpenMP, a simple way to parallelise a code using threads on a shared memory (multicore) computer. We add special comments, parallel constructs, to the code. The compiler reads the comments and generates parallel code.
  - The lecture will cover the most common constructs, some optimization techniques, things to avoid.
  - Most examples will consist of a few lines of code. C, Fortran and some Matlab.
  - The lecture ends with a simple case study, solving a system of stiff odes, parallelising the computation of the Jacobian matrix.

6: A few words about parallel computing

The focus of the Summerschool is on high performance of programs. We can increase the performance using:

- Code optimization on one CPU or core (tomorrow).
- Parallel computing using threads and shared memory (today).
- GPU-programming, special hardware.
- Parallel computing using processes, MPI.

When we optimize a code we are interested in the speedup = time before optimization / time after optimization. In a parallel context, using **P** CPUs/cores, we look at

$$\textbf{\textit{speedup}}(P) = \frac{\textbf{\textit{time on one CPU}}}{\textbf{\textit{time on P CPUs}}}$$

where the time is the wct, wall-clock time and not the total CPU-time (adding the times for all processes/threads together).

7: A few words about parallel computing

Instead of running the parallel program on one CPU it may be more interesting to compare the parallel code with the best sequential (serial) code. Perhaps we had to use a different algorithm to make the code parallel.

We hope to achieve linear speedup, speedup(P) = P.

It is possible to have super linear speedup, speedup(P) > P, this is usually due to better cache locality or decreased paging.

8: A few words about parallel computing

If our algorithm contains a section that is sequential (cannot be parallelised), it will limit the speedup, Amdahl's law.

Let the sequential part be $\boldsymbol{s}$, $\mathbf{0} \leq \boldsymbol{s} \leq \mathbf{1}$ (part wrt time), so the part that can be parallelised is $\mathbf{1} - \boldsymbol{s}$. Hence,

$$\textbf{speedup(P)} = \frac{\mathbf{1}}{\mathbf{s} + (\mathbf{1} - \mathbf{s})/\mathbf{P}} \leq \frac{\mathbf{1}}{\mathbf{s}}$$

regardless of the number of CPUs.

Let n be a measure of problem size (e.g. matrix dimension). Often

$$\mathbf{wct(n,P)} \approx \frac{\mathbf{comp(n)}}{\mathbf{P}} + \mathbf{serial} + \mathbf{P} \cdot \mathbf{comm(n)}$$

so it is not optimal letting $P \longrightarrow \infty$, unless the problem is embarrassingly parallel, no communication.
We are interested in how well the program scales with n and P.
Efficiency is another measure for understanding the balance between computation, communication and synchronisation.

$$\mathbf{efficiency(P)} = \frac{\mathbf{speedup(P)}}{\mathbf{P}}$$

So, if speedup(P)=Pc, efficiency(P) = c. Amdahl's model gives:

$$\mathbf{efficiency(P)} = \frac{\mathbf{1}}{\mathbf{Ps + 1 - s}}$$

so efficiency(1) = 1, efficiency(∞)=0 (if $\boldsymbol{s > 0}$).

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.

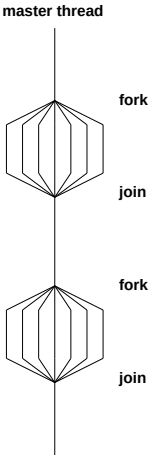
- Fortran version 1.0, 1997, ver. 2.0, 2000.
- C/C++ ver. 1.0 1998, ver. 2.0, 2002.
- Version 2.5, 2005, combined the Fortran and C/C++ specifications into a single one and fixed inconsistencies.
- Version 3.1, 2011, not supported by all compilers.

v2.5 mainly supports data parallelism (SIMD), all threads perform the same operations but on different data.
v3.0 added "tasks", different threads perform different operations.
v4.0 specification under development.
Specifications: **`www.openmp.org`** .
Books: **`http://openmp.org/wp/resources/#Books`**


```
program test
... serial code ...
!$OMP parallel shared(A, n)
... code run i parallel ...
!$OMP end parallel

... serial code ...
!$OMP parallel do shared(b, c)
... code run i parallel ...
!$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.
  `export 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

Parallelising using distributed memory (MPI):

- Requires large grain parallelism to be efficient (process based).
- Large rewrites 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.

- 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.


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

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

// a parallel for loop
#pragma omp parallel for private(i) shared(a, b)
  for (i = 0; i < n; i++) {
    b[i] = sin(1.0e-5 * i);
    a[i] = 1.23 * b[i] + exp(b[i]);
  }
  printf("%f, %f\n", a[0], a[n - 1]);  // the master
  printf("num threads = %d\n",omp_get_num_threads());
```



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

    #pragma omp master
    {
     // number of executing threads
     // max. number of threads that can be started
     // number of available cores
     printf("num threads = %d, max threads = %d, max cpus = %d\n",
            omp_get_num_threads(), omp_get_max_threads(),
            omp_get_num_procs());
    }  // use { } for begin/end
  }
  return 0;
}
```

```
// a parallel for loop
#pragma omp parallel for private(i)
  for(i = ...
```

is short for

```
// a parallel region
#pragma omp parallel private(i)
{
  #pragma omp for  // with a for loop
  for(i = ...
}
```

but the parallel region is more general.

- Use **shared** when a variable is not modified in the loop (read only) or when threads write to different elements in an array
- All variables except the loop-iteration variable are **shared** by default. **default(none)** turns off the default.
- 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 may ask the master thread (thread zero) to print.


```
% icc -openmp omp1.c -lm  use -openmp and optim. flags
% export OMP_NUM_THREADS=1
(setenv OMP_NUM_THREADS 1 in tcsh )
% a.out
1.000000, 1.227759
num threads = 1  Only the master
i_am = 0
num threads = 1, max threads = 1, max cpus = 16

% export OMP_NUM_THREADS=3
% a.out
1.000000, 1.227759
num threads = 1  Only the master
i_am = 0
num threads = 3, max threads = 3, max cpus = 16
i_am = 1  Not in order
i_am = 2
```


- Some systems warn you if # of threads > # of cores/cpus.
- Make no assumptions about the order of execution between threads.
- Output from several threads may be interleaved.
- **ifort -openmp ...**, **icc -openmp ...** Intel
- **gfortran -fopenmp ...**, **gcc -fopenmp ...** GNU
- **pgf90 -mp ...**, **pgcc -mp ...** Portland group
- **path90 -openmp ..**, **pathcc -openmp ..** PathScale

```
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

!$omp parallel do private(i), shared(a, b)
  do i = 1, n
    b(i) = sin(1.0d-5 * i)
    a(i) = 1.23d0 * b(i) + exp(b(i))
  end do
!$omp end parallel do    ! not necessary

  print*, a(1), a(n)     ! only the master
  print*, 'num threads = ', omp_get_num_threads()
```



```
!$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 **}**.

First a silly example:

```
  int a, i;

#pragma omp parallel for private(i) shared(a)
  for (i = 0; i < 1000; i++)
    a = i;   // all threads write to (the same) a
  printf("%d\n", a);
```

Repeated runs may give different values **999**, **874** etc.
Now for a less silly example:

```
  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? Assume thread zero does the first three iterations, thread one the next three etc.

```
thread          computation
  0          a[1]  = a[0] + b[0]
  0          a[2]  = a[1] + b[1]
  0          a[3]  = a[2] + b[2]

  1          a[4]  = a[3] + b[3]
  1          a[5]  = a[4] + b[4]
  1          a[6]  = a[5] + b[5]

  2          a[7]  = a[6] + b[6]
  2          a[8]  = a[7] + b[7]
  2          a[9]  = a[8] + b[8]

  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 (but the threads do not run in parallel):

```
// 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];
```

This may be used in a parallel region but one would not use it in a standalone loop.

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++) {   // different variables
    ...
  }
```

The same variable must occur in all three parts of the loop.
More general types of loops are illegal as well, e.g.

```
  for(;;) {   // 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.


```
  a = (/ 1, 2, 3, 4, 5, 6 /)
 !$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

% 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
    2    3    4    4    5    6   on one thread
    2    3    4    5    5    6   on two threads
```

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

#pragma omp parallel for  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)
```


Use lastprivate when you want to keep the result from the last thread.
(Private variables are not defined outside the parallel region.)

```
  int i, n = 16, v[n]; // Set the v[i] to -1

#pragma omp parallel for   private(i, v)
  for (i = 0; i < n; i++)
    v[i] = omp_get_thread_num();
// Print i, :, followed by all the v[i]

#pragma omp parallel for  lastprivate(i, v)
  for (i = 0; i < n; i++)
    v[i] = omp_get_thread_num();
// Print i, :, followed by all the v[i]

% a.out  On four threads (note, i not initialized)
839015264: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
16: 100 100 -2114056896 32536 26582016 0 26582208 0 0 1 265
 3 3 3 3  last four numbers
```


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).

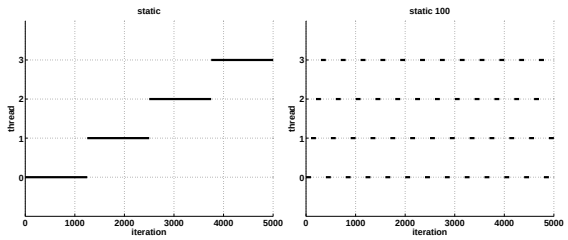
• static
Chunks of iterations are assigned to the threads in cyclic order. Size of default chunk, roughly = n / number of threads.
Low overhead, good if the same amount of work in each iteration.
chunk can be used to access array elements in groups (may be more efficient, e.g. using cache memories in 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
```

The chunk can be set with a variable or an expression as well.
Here is a larger problem, where $n = 5000$, schedule(static), and using four threads.



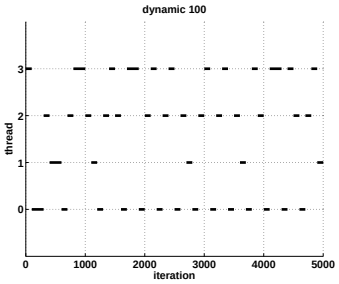
Note that if the chunk size is 5000 (in this example) only the first thread would work, so the chunk size should be chosen relative to the number of iterations.

- 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.

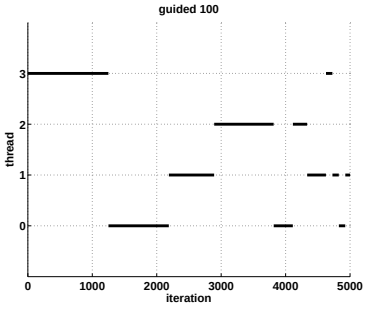
```
!$omp parallel do  schedule(dynamic, chunk) private(k)
```

Here a run with schedule(dynamic,100) (schedule(dynamic), gives a chunks size of one). The amount of works differs between iterations.



- guided

There is also schedule(guided, chunk) assigning pieces of work ($\geq$ 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)  schedule(runtime)
```

```
% export OMP_SCHEDULE=dynamic          bash
% setenv OMP_SCHEDULE "guided,100"     tcsh
% a.out
```

Useful for testing.

Suppose we parallelise m iterations over P processors/cores.
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.

```
!$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. Instead of the optimal speedup P we get:

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

The next example shows this in practice, but first a comment.


```
const int M = 1000, MAX_THREADS = 8;
double time;
int j, k, thr;

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

  time = omp_get_wtime();  // a builtin function
  #pragma omp for schedule(runtime) private(...
  for (j = 1; j <= M; j++)
    for (k = j + 1; k <= M; k++)
      work(...);  // Equal amount of work

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


Here are the times:

#threads	static	static, 10
1	3.83	3.83
2	2.88	1.94
3	2.13	1.30
4	1.68	0.99
5	1.38	0.80
6	1.17	0.67
7	1.02	0.58
8	0.90	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]++;  // how many iters. am I doing?
}
printf("%5.2f, %d %d %d %d\n",
       omp_get_wtime() - time,
       iter[0], iter[1], iter[2], iter[3]);
```



```
static
0.01,  25000000 25000000 25000000 25000000

dynamic, "dynamic,10", "dynamic,100"
15.53, 25611510 25229796 25207715 23950979
1.32,  25509310 24892310 25799640 23798740
0.13,  29569500 24044300 23285700 23100500

guided
0.00,  39831740  5928451 19761833 34477976
```


If several threads try to update the same variable this variable has to be protected so that only one thread at a time can update the variable. It is OK if several threads read the same memory location at the same time. Here is a parallelisation of an inner product computation using a reduction-variable (reduction, many to one).

```
int i, n = 10000;
double x[n], y[n], s;
// Assign values to x and y
...
// s will behave as shared variable,
// but it is protected
s = 0.0;
#pragma omp parallel for  reduction(+: s) shared(n,x,y) privat
for (i = 0; i < n; i++)
  s += x[i] * y[i];  // s must be protected
```


In general: **reduction(operator: variable list)** where operator is **+, *, −, &, |, ^, &&, ||, min, max**.

A reduction is typically specified for statements of the following form (**expr** is of scalar type and does not reference **x**).

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

reduction(operator or intrinsic: variable list)
Valid operators: **+, *, -, .and., .or., .eqv., .neqv.**
and intrinsics: **max, min, iand, ior, ieor** (**iand** is bitwise 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**.

x may be an array in Fortran (vector reduction) but not so in C.

Here an example where we use an intrinsic function in Fortran:

```
! Init. vec(k) = 1.0 / k, k = 1, 2, ..., n

  min_vec = vec(1)  ! important
!$omp parallel do reduction(min: min_vec)  &
!$omp shared(vec) private(k)
  do k = 1, n
    min_vec = min(vec(k), min_vec)
  end do

  print*, 'min_vec = ', min_vec
```

min_vec = vec(1) is important, otherwise we will get an undefined value. Setting **min_vec = -1** gives a minimum of **-1** (in this example).

Now for min in C (not supported by all compilers):

```
  min_vec = vec[0]; // important
#pragma omp parallel for  reduction(min: min_vec) \
        shared(vec)
  for(int k = 0; k < n; k++)
    if ( min_vec > vec[k] )
      min_vec = vec[k];
```

An alternative is

```
  min_vec = min_vec > vec[k] ? vec[k] : min_vec;
```


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 a critical section. A critical section is a piece of code where only one thread at a time is given permission to execute the piece.

```
  double private_s, shared_s = 0;

#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];
    // Only one thread at a time may pass through.
    // Safe update of  shared_s  with  private_s.
    #pragma omp critical
    shared_s += private_s;
  }
```

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

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

  #pragma omp parallel shared(shared_sum, n)
  { // should check return from malloc
    double *private_sum = malloc(n * sizeof(double));
    for(int k = 0; k < n; k++)
      private_sum[k] = ... // compute ...

  // Not too bad with a critical section here.
    #pragma omp critical
      for(int k = 0; k < n; k++)
        shared_sum[k] += private_sum[k];

    free(private_sum);
  } // 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();


   // Each thread computes its own partial_sum.
    for(k = 0; k < n; k++)  // done by all
      S[i_am][k] = ..
```



```
   // At a barrier the threads synchronise.
   #pragma omp barrier  // S must be complete

   // Add the partial sums together
   // (could be stored in S).
    #pragma omp for
    for(k = 0; k < n; k++)
      for(j = 0; j < n_threads; j++)
        shared_sum[k] += S[j][k];   // No conflict.
  } // end parallel
```


Computing the matrix-vector product in two ways (contrived example, use BLAS instead).

```
! Column oriented version
  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** (different threads will write to the same **a(i)**).
May be inefficient since parallel execution is initiated **n** times.
OK if **n** small and **m** large.

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

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


Here is a test, the loops were run ten times.

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.


```
#pragma omp parallel shared(...
{
#pragma omp single   // only ONE thread will execute
      ... code

#pragma omp for nowait  // don't wait, wait is default
    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 // conditional compilation
  C statements ...  Included if we use OpenMP
# endif
} // end of the parallel section
```

Now to Fortran:

```
!$omp parallel shared(...
!$omp single   ! only ONE thread will execute the code
        ... code
!$omp end single

!$omp do
        do ...
        end do
!$omp end do nowait  ! do not 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

!$ code ... conditional compilation
   the space after !$ is important

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


Do not use critical sections and similar constructions too much. This test uses reduction, critical and atomic. $n = 10^7$ using one to four threads.

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

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

#pragma omp parallel for shared(s) private(i)
for (i = 1; i <= n; i++) {
  #pragma omp atomic   // atomic updates a single
  s += sqrt(i);        // variable atomically.
}
```


Here are the times:

#threads	reduction	critical	atomic
1	0.036	0.67	0.19
2	0.020	5.57	0.54
3	0.014	5.56	0.84
4	0.010	5.30	1.14

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

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

```
! 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
```

or shorter

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

Now an example where a function is called from a parallel region. If we have time leftover there will be more at the end of the lecture.

First a definition:

```
  call sub(A, b, x, n)  ! actual parameters
...

  subroutine sub(A, b, x, n)  ! formal parameters
```

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.
In Fortran all variables are passed by reference, so they inherit the data-sharing attributes of the associated actual parameters.
A C-example comes on the next page.


```
// I have chosen int since it gives short lines.
void work(int [], int [], int, int, int  *, int *);
...
int  p_vec[n], s_vec[n], p_val, s_val, p_ref, s_ref;
...
#pragma omp parallel private(p_vec, p_val, p_ref)
                     shared(s_vec, s_val, s_ref)
{
  work(p_vec, s_vec, p_val, s_val, &p_ref, &s_ref);
}
void work(int p_vec[], int s_vec[], int p_val,
          int s_val,   int *p_ref,  int *s_ref)
{
  // p_vec, p_val, p_ref become private
  // s_vec, s_ref become shared
  // s_val becomes private, each thread has its own
  int k;  // becomes private
```



```
  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]);
  }
...
  void work(int a[], int i_am)
  {
    // a[] becomes shared, i_am becomes private
    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.

- Message passing using the "Distributed Computing Toolbox".
- Matlab R2011b (and later) has support for Nvidia's CUDA (Compute Unified Device Architecture), for executing code on the GPU. Requires compute capability ≥ 1.3 (double precision, among other things).
- Threads & shared memory by using the underlying numerical libraries, AMD's ACML (AMD Core Math Library) and Intel's MKL (Math Kernel Library).

We are only going to look at the last point.
Using the Matlab-function **maxNumCompThreads** one can set the number of threads, though in version R2010b it says that **maxNumCompThreads** will be removed in a future version.
Unless you use **matlab -singleCompThread** Matlab uses all cores (and possibly hyperthreading).

Here is a test using one to four threads computing
C = A * B, **x = A \ b** and **l = eig(A)**
(**n** is the order of the matrices).

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 multiplication, than for **eig**, which seems reasonable. This method can be used to speed up the computation of elementary functions as well.

MKL and ACML can be used from Fortran and C as well. Here $\boldsymbol{Ax} = \boldsymbol{b}$ is solved using **dgesv** from Lapack (a part of ACML and MKL).

```
call dgesv(n, nrhs, A, lda, ipiv, b, ldb, info)
```

Compile and link with the library, and set **LD_LIBRARY_PATH**.
Performance on 1 to 5 threads, $\boldsymbol{A}$ is an unsymmetric $\boldsymbol{n} \times \boldsymbol{n}$-matrix.

n	1	2	3	4	5
2000	0.37	0.21	0.16	0.13	0.12
4000	2.7	1.4	1.0	0.8	0.7
8000	20.0	10.3	7.1	5.5	4.9
16000	154.2	82.9	55.5	42.6	35.0

$\boldsymbol{y'(t) = f(t, y(t)),\ y(0) = y_0,\ y : \mathbb{R} \to \mathbb{R}^n}$,
we assume $\boldsymbol{f(t, y)}$ is expensive to evaluate.

LSODE (Livermore Solver for ODE, Alan Hindmarsh) from
www.netlib.org .

BDF routines; Backward Differentiation Formulas.
Implicit methods: $\boldsymbol{t_k}$ present time, $\boldsymbol{y^{(k)}}$ approximation of $\boldsymbol{y(t_k)}$.

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

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

LSODE is adaptive (can change both the timestep $\boldsymbol{h}$ and the order).

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

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

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 a 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.
Compare with the scalar case: $g'(z) \approx (g(z+\delta) - g(z))/\delta$.
A modified Newton is used, so one Jacobian and one factorization is used every timestep.

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

Major costs in LSODE:

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

What speedup can we expect?

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

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

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?

- Switch from Linpack, used in LSODE, to Lapack.
- Use a parallel library like ACML or MKL.

After having searched LSODE (Fortran 66):

```
c if miter = 2, make n calls to f to approximate j.
      j1 = 2                          used for indexing
      do 230 j = 1,n                  for each column in J
        yj = y(j)                     save value
        r = dmax1(srur *dabs(yj),r0/ewt(j))   = δ_j
        y(j) = y(j) + r               perturb
        fac = -h10/r                  = −h/δ_j
        call f (neq, tn, y, ftem)     tn = t_{k+1}, ftem new f-value
        do 220 i = 1,n                wm(2...) stores the Jacobian
220      wm(i+j1) = (ftem(i) - savf(i)) *fac form diff. quot.
        y(j) = yj                     restore perturbed value
        j1 = j1 + n                   next column
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)  Newton iter.
```

Simple to parallelise the loop. Can forget the remaining 2500 lines.
The parallel version: important to look at how each variable is used.

- **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,
c$omp+        n, neq, tn)  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)
```


Calling a function, containing OpenMP-directives, from a parallel region.

lexical extent of the parallel region

```
int main()
{
 ...
 #pragma omp parallel ...
 {
  #pragma omp for ...
  ...; work(...); ...
 }
 ...
}

void work(...)
 #pragma omp for (...)  {
  ...
 }
 ...
}
```

dynamic extent of the parallel region

```
int main()
{
 ...
 #pragma omp parallel ...
  {
  #pragma omp for ...
   ...; work(...); ...
  }
 ...
}

void work(...)
 #pragma omp for (...)  {
  ...
 }
 ...
}
```


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 code and that we have three threads:

```
int k, i_am;
char f[] = "%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, i_am, omp_get_thread_num(), k);

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

#pragma omp parallel for private(k)
for (k = 1; k <= 6; k++)       // LOOP 3
  printf(f, 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, **OMP_NESTED** to **TRUE**, or use **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.

First loop			Second loop			Third loop		
i_am	omp()	k	i_am	omp()	k	i_am	omp()	k
1	1	3	0	0	1	1	0	1
1	1	4	1	1	1	1	0	2
2	2	5	1	1	2	1	2	5
2	2	6	2	2	1	1	2	6
0	0	1	0	0	2	1	1	3
0	0	2	0	0	3	1	1	4
			1	1	3	2	0	1
			1	1	4	2	0	2
			1	1	5	2	1	3
			1	1	6	2	1	4
			2	2	2	2	2	5
			2	2	3	2	2	6
			2	2	4	0	0	1
			2	2	5	0	0	2
			2	2	6	0	1	3
			0	0	4	0	1	4
			0	0	5	0	2	5
			0	0	6	0	2	6

- Optimize 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 optimization 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 to avoid synchronisation (**barrier**, **critical**, **atomic**)
  - try to avoid **single**-sections
  - examine the different ways the loop can be parallelised
  - do not forget single-CPU optimization (cache locality)
  - choose a suitable schedule