Parallel Computing

Programming with OpenMP

Kevin Dunseath

Computational Science and Applications

2017–2018

- Books:
  - ”Parallel Programming in OpenMP”, Chandra *et al*, Morgan Kaufmann Publishers (October 2000)
  - ”Using OpenMP”, Chapman *et al*, MIT Press (2008)
- Websites:
  - http://openmp.org
  - http://www.idris.fr/formations/openmp/

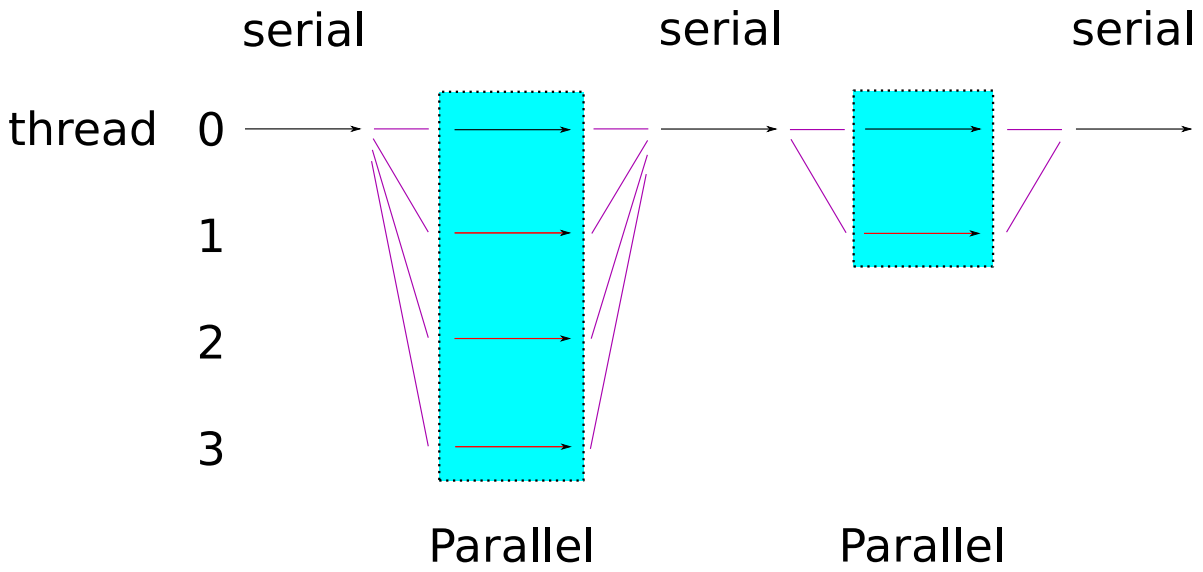
Version 4.0 of the OpenMP specification was released in July 2013

OpenMP (Open Multi-processing)

- an Application Program Interface (API) that allows multi-threaded parallelism on shared memory computers
- available in C, C++, Fortran, . . . ,
  and on many different types of machine
- consists of compiler directives, runtime library routines and environment variables

Parallel programming using threads

- Program begins as a single process or master thread
- Master thread forks (splits) into a number of independent tasks or threads that run simultaneously



- The runtime environment allocates threads to processors
- The number of threads is specified by the environment variable `OMP_NUM_THREADS` or by calls to the runtime library function `omp_set_num_threads()`
- Regions of the program to be run in parallel are indicated using a preprocessor directive; this instructs the operating system to create threads before the region is executed
- Each thread has a unique integer identifier, given by `omp_get_thread_num()`
  The master thread has an id of 0

- Each thread has its own stack, used for storing ”stack frames” associated with subroutine or function calls: threads may call functions without interfering with stack frames of other threads
- Other program variables may be shared by all threads, or private copies may be made for each thread: the private copies are stored on the thread’s own stack
- When execution of the parallel region has completed, the threads are killed and execution continues in the master thread

Worksharing

Threads divide up or share the work that needs to be done

- loop sharing, where different threads execute different loop iterations: care is needed to ensure each iteration is independent of the others (dependency elimination)
- sections, where different threads execute different independent regions of code, one region per thread
- orphaning, where a function is called several times with different data: each thread performs one function call

Advantages:

- Simpler to implement than message passing
- Can parallelise one bit of the program at a time
- Same code for both serial and parallel applications: serial code not modified

Disadvantages:

- Only runs on homogeneous, shared-memory machines
- Requires a compiler and libraries that support threading
- Scalability is limited (memory bus may become saturated)
- Thread creation takes time, and can severely impact a program's performance if not done carefully

Defining a parallel region: OpenMP directives

General form: sentinal directive [clause . . .]

Fortran:
```
!$omp ...
c$omp ...
*$omp ...
```
{ fixed form source (Fortran 77)

C/C++:
```
#pragma omp ...
```

Fortran is case insensitive, C/C++ are case sensitive

Compilation:

gcc -fopenmp . . .
gfortran -fopenmp . . .

ifort -openmp . . .
. . . -lmkl_intel_thread
(MKL multithread routines)

In Fortran

- `use OMP_LIB` provides access to OMP library functions
- lines starting with `!$` are ignored unless the compiling with OpenMP support
- a parallel region is defined by

```
!$omp parallel
....
!$omp end parallel
```

In C/C++

- `#include "omp.h"` declares OMP library functions
- the symbol `_OPENMP` is defined when compiling with OpenMP support: this can be used to determine if particular lines of code should be compiled or not
- a parallel region is defined by

```
#pragma omp parallel
{
....
}
```

```
program hello_omp
!$ use omp_lib
implicit none
integer :: num_threads, rank
!$omp parallel        ← threads created here
!$ num_threads = omp_get_num_threads()
!$ rank = omp_get_thread_num()
!$ write (6,*) "Hello from thread ", rank, &
!$              " out of ", num_threads
!$omp end parallel    ← threads destroyed here
stop
end program hello_omp

> gfortran hello_omp.f90
> ./a.out
> gfortran -fopenmp hello_omp.f90
> ./a.out
 Hello from thread           1  out of           2
 Hello from thread           1  out of           2
```

By default `num_threads` and `rank` are shared

```
program hello_omp
!$ use omp_lib
implicit none
integer :: num_threads, rank
!$omp parallel private (rank)
!$ num_threads = omp_get_num_threads()
!$ rank = omp_get_thread_num()
!$ write (6,*) "Hello from thread ", rank, &
!$              " out of ", num_threads
!$omp end parallel
stop
end program hello_omp
```

```
> gfortran -fopenmp hello_omp.f90
> ./a.out
 Hello from thread           0  out of           2
 Hello from thread           1  out of           2
>
```

In C:

```
#include <stdio.h>

#ifdef _OPENMP
#include "omp.h"
#endif

int main () {
int num_threads, rank;
#ifdef _OPENMP
#pragma omp parallel private (rank)
{
   num_threads = omp_get_num_threads();
   rank = omp_get_thread_num();
   printf ("Hello from thread %d out of %d\n",
            rank,num_threads);
}
#endif
    return 0;
}
```

In C++:

```
#include <cstdio>

#ifdef _OPENMP
#include "omp.h"
#endif

int main () {
#ifdef _OPENMP
#pragma omp parallel
{
   int num_threads = omp_get_num_threads();
   int rank = omp_get_thread_num();
   printf ("Hello from thread %d out of %d\n",
            rank,num_threads);
}
#endif
    return 0;
}
```

The scope of `num_threads` and `rank` is limited to the block in which they are declared: they are local and private to each thread

Parallel regions

- Inside a parallel region, the variables are shared by default
  This can be changed using the `default()` clause:

  ```
  !$omp parallel default(private)
  ```

  In Fortran, the possible values are `private`, `firstprivate`, `shared` or `none`
  In C/C++, the possible values are `shared` or `none`
- Local variables declared within a block in C++ are private
- Inside a particular parallel region, all threads execute the same code
- At the end of a parallel region, execution is synchronised: the master thread continues executing only when all threads have reached the end of the parallel region
- There must no branching into or out of the parallel region

Parallel regions (2)

- Private variables are stored on the thread's personal stack: they are therefore not initialised
- Private variables may be initialised to their value just before the start of the parallel region using the clause `firstprivate()`:

```
int base = 100;
#pragma omp parallel default(none) firstprivate(base)
{
    int rank = omp_get_thread_num();
    base += rank;
    printf("Thread %d: base = %d\n",rank,base);
}
```

```
private(base):                  firstprivate(base):
Thread 0:  base = 8388608       Thread 0:  base = 100
Thread 1:  base = 1             Thread 1:  base = 101
```

Loop-level parallelism

A simple example:

```
!$omp parallel do
do i = 1, n
   z(i) = a*x(i) + y(i)
end do
!$omp end parallel do
```

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    z[i] = a*x[i] + y[i];
}
```

- Each iteration of the loop is independent of the others: different iterations can be executed simultaneously
- Slave threads are created at the beginning of the loop, and are destroyed at the end
- The iterations are distributed over the available threads
- The loop indices are private to each thread
- Execution continues only after all threads have finished

Loop-level parallelism (2)

The directive is a combination of

- a directive declaring a parallel region (`!$omp parallel`)
- a directive declaring a parallel loop (`!$omp do`)

These can be used separately, allowing several parallel loops to be used in one parallel region:

```
!$omp parallel
!$omp do
do i = 1, n
   x(i) = 0.0;
end do
!$omp end do nowait
!$omp do
do j = 1, 3*n-1
   y(j) = sin(pi*j/180.0);
end do
!$omp end do
!$omp end parallel
```

This is more efficient than using `!$omp parallel do` twice: the slave threads are created once at the first `!$omp parallel` and are used for both loops

Using `!$omp end do nowait` allows threads to continue executing without waiting for other to finish

Loop-level parallelism (3)

Iterations are assigned to threads in continous chunks

The chunk size is the number of iterations assigned to a thread

The distribution of iterations over available threads may be controlled using the `schedule` clause:

```
schedule(type[, chunk])
```

- `static`: the choice of which thread executes which iteration is simply a function of the iteration number and the number of threads. Each thread performs the iterations assigned to it at the beginning of the loop. For example, if there are 100 iterations and 4 threads, each thread will perform 25 iterations. This is efficient if each iteration contains the same amount of work

Loop-level parallelism (4)

- `dynamic`: the assignment of iterations to a thread can vary from one execution to the next. Not all iterations are assigned at the beginning of a loop: a thread may ask for more iterations after it has completed those already assigned to it. This is more efficient if some iterations take longer than others, but has a higher overhead
- `guided`: the first chunk is a pre-defined size, the size of each following chunk decreases exponentially until a minimum size is reached. The chunks are distributed dynamically
- `runtime`: the type of scheduling is determined from the environment variable `OMP_SCHEDULE`

The choice of scheduling and chunk size can have a critical effect on performance

Reduction loops

- A reduction clause performs a reduction operation on the each of the (shared) variables in its list
- Each thread has a private copy of the reduction variable
- The reduction is performed on all private copies of the reduction variable, and the result is stored in the shared variable
- Valid reduction operations include arithmetic and logical operations, finding the maximum or minimum of a set of data

```
sum = 0.0;
prod = 1.0;
#pragma omp parallel for reduction(+:sum) reduction(*:prod)
for (int i=0; i<n; i++) {
    sum += a[i];
    prod *= a[i];
}
```

Private variable initialisation and finalisation

- Private variables stored in thread's own stack
  - $\Rightarrow$ private copies are not initialised
  - $\Rightarrow$ private copies are lost at end of parallel loop
- `firstprivate` clause: at start of a parallel loop, each private copy of a variable is initialised to the value stored by the master thread
- `lastprivate` clause: at the end of a parallel loop, the value of the private variable corresponding to the last iteration is written back to the copy stored by the master thread

```
x = 42;
#pragma omp parallel for firstprivate (x) lastprivate (y)
for (int i=0; i<n; i++) {
    y = x*a[i];
}
printf("y = %d\n",y); // should print 42*a[n-1]
```

Non-iterative work sharing: parallel sections

- A serial program may contain a sequence of different tasks that do not depend on the results of previous tasks
- It may be useful to assign such tasks to different threads
- This can be achieved using parallel sections

```
!$omp parallel sections ...
!$omp section
 code for first section
!$omp section
 code for second section
!$omp end sections
```

```
#pragma omp parallel sections ...
{
  #pragma omp section
     block
  #pragma omp section
     block
}
```

Allowed clauses are

`private, firstprivate, lastprivate, reduction`

```
double a[n];
double w[m];
double x,y;
#pragma omp parallel sections
{
#pragma omp section        ← executed by one thread
  {
    for (int i=0; i<n; i++)
        a[i] = (double) rand()/RAND_MAX;
    x = a[0];
    y = a[n-1];
  }
#pragma omp section        ← executed by one thread
  {
    w[0] = 1.0;
    for (int j=1; j<m-1; j += 2) {
        w[j] = 4.0;
        w[j+1] = 2.0;
    }
    w[m-1] = 1.0;
  }
}
```

Assigning work to a single thread

- Sometimes a parallel region contains tasks that should be executed by only one thread
- Such tasks can be identified using the `single` directive

```
!$omp single ...
  block of code to be executed
  by only one thread
!$omp end single
```

```
#pragma omp single ...
{
  block of code to be executed
  by only one thread
}
```

- The single directive may be followed by a `private` or `firstprivate` clause
- The task is usually executed by the thread that arrives first at this directive

- There is an implicit barrier at the end of a `single` region: other threads wait for the thread executing the region to finish before continuing the execution of the parallel region. Adding `nowait` after `!$omp end single` (Fortran) or after `#pragma omp single` (C/C++) removes this barrier
- The `single` directive is typically used when there is some I/O to be performed

The `master` directive

- The `master` directive is similar to `single`, except that the execution of the specified region is performed by the master thread (rank = 0)
- The `master` directive accepts no clauses
- There is no implicit barrier: all threads except the master thread skip past this block of code
- Like `single`, this directive can be used to ensure all I/O is performed by the master thread itself

Synchronisation

- Barrier synchronisation: execution halts at the barrier until all threads have arrived at this point. Most parallel constructs have an implicit barrier at the end which can be removed using a `nowait` clause The user can introduce a barrier with the directive `!$omp barrier` or `#pragma omp barrier`
- Data access synchronisation: several threads may need to access or modify a shared variable, and the order in which they do so may affect the final result. It is often necessary to coordinate access to shared variables by multiple threads in order to ensure correct results.

Example:

```
double max = -DBL_MAX;
#pragma omp parallel for
for (int i=0; i<n; i++) {
    max = (max > a[i]) ? max : a[i];
}
printf ("max = %f\n", max);
```

Suppose there are two threads and that currently `max=10`

Suppose further that in the first thread `a[i]=12` while in the second thread `a[i']=11`

Thread 0	Thread 1
`read a[i] (12)`	`read a[i'] (11)`
`read max (10)`	`read max (10)`
`max > a[i] (10 > 12)`	`max > a[i'] (10 > 11)`
`store max = 12`	`store max = 11`

If thread 1 modifies `max` after thread 0, the wrong value is stored

Synchronisation (2)

- Data race: one thread tries to read a shared variable when another is modifying it (concurrent access): which value will be read?
- Access to shared data must be coordinated (synchronised) to ensure correct results
- Some data races can be removed by making private copies of the shared variable. The previous example could also be treated as a `reduction`
- If all concurrent accesses to a shared variable are reads, no synchronisation is required

```
!$omp parallel do shared(a,b)
do i = 1, n
   a(i) = a(i) + b
end do
!$omp end parallel do
```

Synchronisation (3)

Mutual exclusion synchronisation (thread safety)

When several threads need access to a shared variable or data structure, this guarantees that only one thread at a time has access to the data, and that the accesses are interleaved

Critical sections

```
!$omp critical
block of code
!$omp end critical
```

```
#pragma omp critical
{
block of code
}
```

- When it arrives at a `critical` directive, a thread waits until no other thread is executing inside the critical section
- It then has exclusive access to the critical section
- When it has finished inside the critical section, the thread releases its exclusive access and continues execution beyond the critical section
- Only one critical section is allowed to execute at one time anywhere in the program

Atomic directive

- Like `critical`, the `atomic` directive guarantees that a shared variable can be read or modified by only one thread at a time
- Unlike `critical`, it applies only to the line or instruction following it
- It is implemented by a set of hardware synchronisation primitives resulting in higher performance

Atomic directive (2)

- The instructions following `atomic` are restricted to the forms

```
!$omp atomic
x = x operator expression

#pragma omp atomic
x <binary operator>= expression
```

- `operator` represents most arithmetic and logical operators
- `expression` is a scalar expression that does not use `x`

Atomic directive (3)

`atomic` is often used to update counters

```
int positive = 0;
int negative = 0;
#pragma omp parallel for
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        if (a[i][j] > 0.0) {
#pragma omp atomic
            positive++;
        } else if (a[i][j] < 0.0) {
#pragma omp atomic
            negative++;
        }
    }
}
```

Parallel tasks

- So far, all worksharing constructs we have seen require knowledge of how much work is to be done (loop size, number of parallel regions)
- They cannot be used to parallelize, for example,

```
std::ifstream input;
input.open("input.dat");
int n;
while (input>>n) {
    doSomething(n);
}
```

```
Node node;
while (node) {
    doSomething(node);
    node = node->next();
}
```

- The directive `#pragma omp task` allows such constructs to be run in parallel

Parallel tasks (2)

- A task consists of a set of instructions to be executed, and the data to be treated
- A task is specified by the directive `#pragma omp task`
  It is assumed that each task can be executed independently of the others
- Tasks are placed in a queue; the runtime system then decides when to execute each task
  Execution may be immediate or deferred
- Execution may be synchronized by adding a `taskwait` clause

Parallel tasks (3)

```
std::ifstream input;
input.open("input.dat");
int n;
#pragma omp parallel                          ← Create threads
{
#pragma omp single nowait                     ← One thread reads data
{                                               and adds task to queue
    while (input>>n) {
#pragma omp task firstprivate(n)              ← Task definition
{
        doSomething(n);
}
    }
}
}                                             ← Destroy threads
```

One thread generates the tasks
the other threads execute the tasks

Parallel tasks (4): data scope

Example: a very inefficient way to evaluate Fibonacci numbers

```
long fibonacci (int n) {
    if (n<2) return n;
    long fib1, fib2;
#pragma omp task
{
    fib1 = fibonacci(n-1);
}
#pragma omp task
{
    fib2 = fibonacci(n-2);
}
#pragma omp taskwait
    return fib1+fib2;
}
```

n is private in both tasks

fib1 is now private to task

fib2 is now private to task

Must declare `fib1` and `fib2` as `shared`

Parallel tasks (4): data scope

Example: a very inefficient way to evaluate Fibonacci numbers

n is private in both tasks

```
long fibonacci (int n) {
    if (n<2) return n;
    long fib1, fib2;
#pragma omp task shared(fib1)
{
    fib1 = fibonacci(n-1);
}
#pragma omp task shared(fib2)
{
    fib2 = fibonacci(n-2);
}
#pragma omp taskwait
    return fib1+fib2;
}
```

Must declare `fib1` and `fib2` as `shared`

Dependencies

- When parallelising loops it is essential that the results are the same as when the loop is executed by only one thread (serial execution)
- This implies that the loop contains no dependencies: one iteration does not require the results of another
- Some dependencies can be avoided or removed, allowing the loop to be run correctly in parallel

If one statement reads data from a location in memory, and another writes to that location, there is a dependency and hence a data race condition

```
for (int i=1; i<n; i++)
    a[i] += a[i-1];
```

Each iteration modifies an element of `a` that is accessed by the next iteration

Detecting dependencies

- Each iteration is executed in parallel, but within a single iteration the instructions are executed sequentially
- Dependencies are important between statements executed in different iterations: *loop carried dependencies*
- If a variable is only accessed and never modified within the loop, there are no dependencies
- A loop can be parallelised if
  - all assignements are to arrays
  - each element is modified by at most one iteration
  - no iteration reads elements modified by any other iteration

Can we find two different values of the loop index, `i` and `i'`, for which `i` modifies an element of an array and `i'` reads or modifies the same element? If so, there is a dependence

```
for (int i=0; i<n-1; i++)
    a[i] += a[i+1];
```

```
for (int i=1; i<n; i+=2)
    a[i] += a[i-1];
```

```
for (int i=0; i<n/2; i++)
    a[i] += a[i+n/2];
```

```
for (int i=0; i<=n/2; i++)
    a[i] += a[i+n/2];
```

```
x = 0
for (int i=0; i<n; i++) {
    if (switch_x(i)) x = new_x(i);
    a[i] = x;
}
```

Classifying dependencies using dataflow

Let S_1, S_2 be two statements in a loop, with S_1 performed before S_2 during serial execution. Suppose both statements read from or write to the same location in memory

S_1	write	read	write
S_2	read	write	write
type	flow	anti	output

- **flow** dependencies are true dependencies, but can be parallelised in some situations
- **anti** dependencies can be removed by giving each iteration a private copy of the memory location initialised with the correct value
- **output** dependencies can be removed by making a private copy of the memory location and copying back the value of the final iteration to the shared memory location

Removing anti dependencies

S_1 reads a data location, S_2 writes to the same location

Solution: give each thread a separate copy of the data in the memory location, taking care that S_1 reads the correct value

Serial version:

```
for (int i=0; i<n-1; i++) {
    x = 0.5*(b[i] + c[i]);
    a[i] = a[i+1] + x;
}
```

Parallel version:

```
#pragma omp parallel for shared(a,akeep)
for (int i=0; i<n-1; i++)
    akeep[i] = a[i+1];
#pragma omp parallel for shared(a,akeep) private(x)
for (int i=0; i<n-1; i++) {
    double x = 0.5*(b[i] + c[i]);
    a[i] = akeep[i] + x;
}
```

Removing output dependencies

S_1 and S_2 both write to the same memory location: we must ensure that after the loop the correct final value is stored

Serial version:

```
for (int i=0; i<n; i++) {
    x = 0.5*(b[i] + c[i]);
    a[i] += x;
    d[2] = 2*x;
}
y = x + d[1] + d[2];
```

Parallel version:

```
double d2;
#pragma omp parallel for shared(a) lastprivate(x,d2)
for (int i=0; i<n; i++) {
    x = 0.5*(b[i] + c[i]);
    a[i] += x;
    d2 = 2*x;
}
d[2] = d2;   y = x + d[1] + d[2];
```

Removing flow dependencies

Reduction:

```
double x = 0.0;
for (int i=0; i<n; i++)
    x += a[i];
```

```
double x = 0.0;
#pragma omp parallel for \
 reduction(+:x)
for (int i=0; i<n; i++)
    x += a[i];
```

Loop skewing:

```
for (int i=1; i<n; i++){
    b[i] += a[i-1];
    a[i] += c[i];
}
```

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

Removing flow dependencies (2)

Scalar expansion and loop fissioning:

Serial version:

```
for (int i=0; i<n; i++){
    y += a[i];
    b[i] = (b[i] + c[i])*y;
}
```

Parallel version:

```
ylist[0] = y + a[0];
for (int i=1; i<n; i++)
    ylist[i] = ylist[i-1] + a[i];
y = ylist[n-1];
#pragma omp parallel for shared(b,c,ylist)
for (int i=0; i<n; i++)
    b[i] = (b[i] + c[i])*ylist[i];
```

Removing dependencies

- It is always possible to remove anti and output dependencies
- Flow dependencies may sometimes be removed
- The price to pay is usually the introduction of extra variables, temporary arrays, and perhaps splitting single loops into several simpler loops, some of which can be parallelised
- Must check that the extra memory and computational costs do not exceed the gain in speed from parallelising the loop

Performance

- Parallel regions must have sufficient work to compensate for the overhead of creating threads, work sharing, synchronisation, ...
- OpenMP provides an `if` clause that specifies a minimum loop length which should be run in parallel:

```
#pragma omp parallel for if (n >= 800)
for (int i=0; i<n; i++) {...}
```

- To maximise the amount of work, outermost loops should be run in parallel
- Each thread should perform the same amount of work. If the time taken for an iteration varies significantly, try using `schedule(dynamic)` or `schedule(guided)`. The size of the chunk of iterations distributed to each thread can also have an effect on performance

Performance (2)

- Some approaches to parallelising loops may conflict with guidelines for good performance of serial execution, in particular cache locality. The user must experiment to see if the gain through parallelisation compensates the loss through poor cache locality
- As far as possible, create a few large parallel regions rather than several small regions, to minimise to number of times threads have to be created and destroyed, memory allocated, ...
- Synchronisation should be kept to a minimum
- Avoid large critical regions

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

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

Performance (3)

The wall-clock time taken by a parallel region may be measured using the function `omp_get_wtime`:

```
double t0 = omp_get_wtime();
#pragma omp parallel ...
{
  ...
}
double t1 = omp_get_wtime();
printf("Time taken was %f seconds\n",t1-t0);
```