Parallel Computing and OpenMP Tutorial

Shao-Ching Huang

IDRE High Performance Computing Workshop

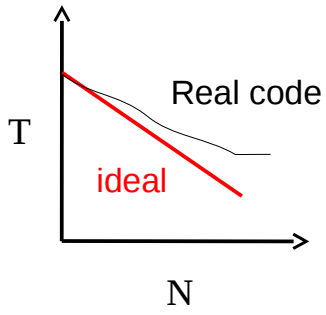
2013-02-11

Why Parallel Computing?

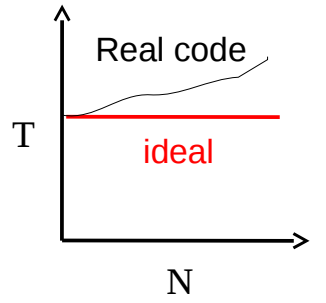
- Bigger data
  - High-res simulation
  - Single machine too small to hold/process all data
- Utilize all resources to solve one problem
  - All new computers are parallel computers
  - Multi-core phones, laptops, desktops
  - Multi-node clusters, supercomputers

Parallel Scalability

- Strong scaling
  - fixed the global problem size
  - local size decreases as N is increased
  - ideal case: T*N=const (linear decay)



- Weak scaling
  - fixed the local problem size (per processor)
  - global size increases as N increases
  - ideal case: T=const.



T(N) = wall clock run time
N = number of processors

Identify Data Parallelism – some typical examples

- "High-throughput" calculations
  - Many independent jobs
- Mesh-based problems
  - Structured or unstructured mesh
  - Mesh viewed as a graph – partition the graph
  - For structured mesh one can simply partition along coord. axes
- Particle-based problems
  - Short-range interaction
    - Group particles in cells – partition the cells
  - Long-range interaction
    - Parallel fast multipole method – partition the tree

Portal parallel programming – OpenMP example

- OpenMP
  - Compiler support
  - Works on ONE multi-core computer

Compile (with openmp support):

```
$ ifort -openmp foo.f90
```

Run with 8 “threads”:

```
$ export OMP_NUM_THREADS=8
$ ./a.out
```

Typically you will see CPU utilization over 100% (because the program is utilizing multiple CPUs)

What is OpenMP?

- API for shared-memory parallel programming
  - compiler directives + functions
- Supported by mainstream compilers – portable code
  - Fortran 77/9x/20xx
  - C and C++
- Has a long history, standard defined by a consortium
  - Version 1.0, released in 1997
  - Version 2.5, released in 2005
  - Version 3.0, released in 2008
  - Version 3.1, released in 2011
- http://www.openmp.org

Which OpenMP version do I have?

GNU compiler on my desktop:

$ g++ --version
g++ (Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5

$ g++ version.cpp –fopenmp
$ a.out
version : 200805

Intel compiler on Hoffman2:

$ icpc --version
icpc (ICC) 11.1 20090630

$ icpc version.cpp -openmp
$ a.out
version : 200805

```
#include <iostream>
using namespace std;
int main()
{
  cout << "version : " << _OPENMP << endl;
}
```

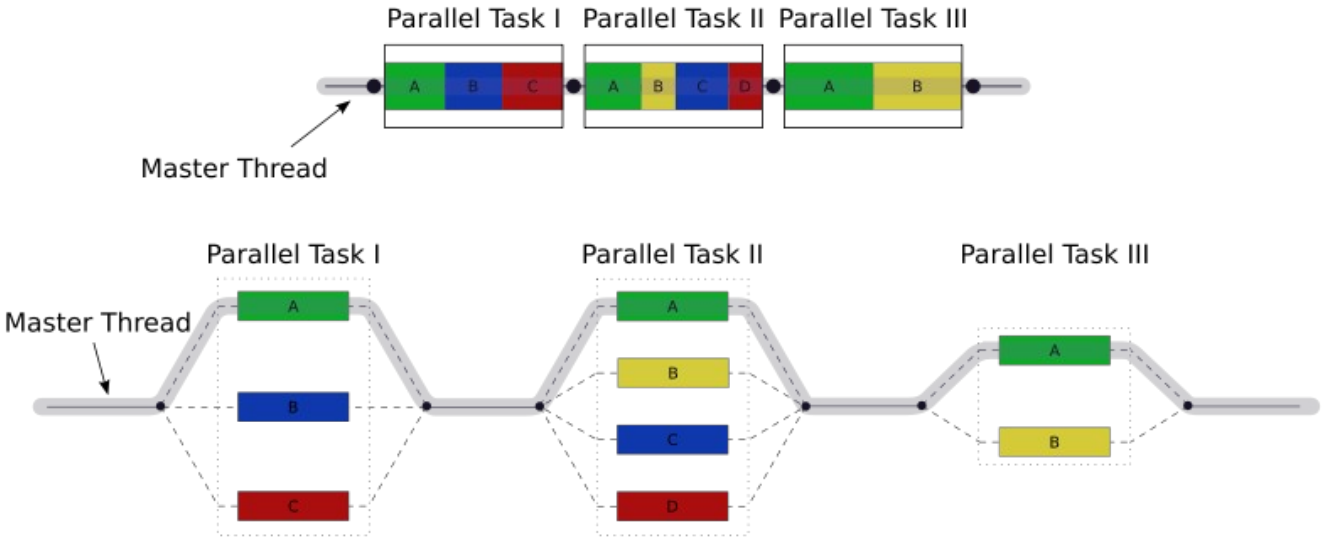
http://openmp.org

Version	Date
3.0	May 2008
2.5	May 2005
2.0	March 2002

Elements of Shared-memory Programming

- Fork/join threads
- Synchronization
  - barrier
  - mutual exclusive (mutex)
- Assign/distribute work to threads
  - work share
  - task queue
- Run time control
  - query/request available resources
  - interaction with OS, compiler, etc.

OpenMP Execution Model



☞ We get speedup by running multiple threads simultaneously.

Source: wikipedia.org

Parallel Region in “main” Program

- Main program is “sequential”
- subroutines/functions are parallelized

```
void main()
{
  #pragma omp parallel
    {
        i = some_index;
        foo(i);
    }
}
```

```
void foo(int i)
{
  // sequential code
}
```

Parallel Region in Subroutines

- Main program is “sequential”
- subroutines/functions are parallelized

```
int main()
{
   foo();
}
```

```
void foo()
{
  #pragma omp parallel
    {
       // some fancy stuff here
     }
}
```


saxpy operation (C)

$$y \leftarrow ax + y$$

Sequential code

```
const int n = 10000;
float x[n], y[n], a;
int i;

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

OpenMP code

```
const int n = 10000;
float x[n], y[n], a;
int i;

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

gcc saxpy.c

gcc saxpy.c -fopenmp

Enable OpenMP support

saxpy operation (Fortran)

$$y \leftarrow ax + y$$

Sequential Code

```
integer, paramter :: n=10000
real :: x(n), y(n), a
Integer :: i

do i=1,n
   y(i) = a*x(i) + y(i)
end do
```

OpenMP code

```
integer, paramter :: n=10000
real :: x(n), y(n), a
integer :: i

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

gfortran saxpy.f90

gfortran saxpy.f90 -fopenmp

Enable OpenMP support

Private vs. shared – threads' point of view

- Loop index "i" is **private**
  - each thread maintains its own "i" value and range
  - private variable "i" becomes undefined after "parallel for"
- Everything else is **shared**
  - all threads update y, but at different memory locations
  - a,n,x are read-only (ok to share)

```
const int n = 10000;
float x[n], y[n], a = 0.5;
int i;
#pragma omp parallel for
for (i=0; i<n; i++) {
   y[i] = a * x[i] + y[i];
}
```

Nested loop – outer loop is parallelized

```
#pragma omp parallel for
  for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        //… do some work here
    } // i-loop
  } // j-loop
```

```
!$omp parallel do
 do j=1,n
  do i=1,n
      !… do some work here
  end do
 end do
```

- By default, only j (the outer loop) is private
- But we want both i and j to be private, i.e.
- Solution (overriding the OpenMP default):

```
#pragma omp parallel for private(i)
```

```
!$omp parallel do private(i)
```

☞ j is already private by default

OpenMP General Syntax

- Header file

  #include <omp.h>

- Parallel region:

☞ Clauses specifies the precise "behavior" of the parallel region

C/C++

```
#pragma omp construct_name [clauses…]
{
    // … do some work here
} // end of parallel region/block
```

Fortran

```
!$omp construct_name [clauses…]
    !… do some work here
!$omp end construct_name
```

- Environment variables and functions (discussed later)

Parallel Region

- To fork a team of N threads, numbered 0,1,..,N-1
- Probably the most important construct in OpenMP
- Implicit barrier

C/C++

```
//sequential code here (master thread)

#pragma omp parallel [clauses]
{
   // parallel computing here
   // …
}

// sequential code here (master thread)
```

Fortran

```
!sequential code here (master thread)

!$omp parallel [clauses]
   ! parallel computing here
   ! …
!$omp end parallel

! sequential code here (master thread)
```


Clauses for Parallel Construct

C/C++ `#pragma omp parallel clauses, clauses, …`

Fortran `!$omp parallel clauses, clauses, …`

Some commonly-used clauses:

- shared
- nowait
- if
- reduction
- copyin
- private
- firstprivate
- num_threads
- default

Clause “Private”

- The values of **private** data are undefined upon entry to and exit from the specific construct
- To ensure the last value is accessible after the construct, consider using “lastprivate”
- To pre-initialize private variables with values available prior to the region, consider using “firstprivate”
- Loop iteration variable is private by default

Clause “Shared”

- Shared among the team of threads executing the region
- Each thread can read or modify shared variables
- Data corruption is possible when multiple threads attempt to update the same memory location
  - Data race condition
  - Memory store operation not necessarily atomic
- Code correctness is user’s responsibility

nowait

C/C++

```
#pragma omp for nowait

// for loop here
```

```
#pragma omp for nowait
...
```

Fortran

```
!$omp do
! do-loop here
!$omp end do nowait
```

```
!$omp do
! … some other code
```

In a big parallel region

- This is useful inside a big parallel region
- allows threads that finish earlier to proceed without waiting
  - More flexibility for scheduling threads (i.e. less synchronization – may improve performance)

If clause

- if (*integer expression*)
  - determine if the region should run in parallel
  - useful option when data is too small (or too large)
- Example

C/C++

```
#pragma omp parallel if (n>100)
{
   //…some stuff
}
```

Fortran

```
!$omp parallel if (n>100)

   //…some stuff

!$omp end parallel
```

Work Sharing

- We have not yet discussed how work is distributed among threads...
- Without specifying how to share work, all threads will redundantly execute all the work (i.e. no speedup!)
- The choice of work-share method is important for performance
- OpenMP work-sharing constructs
  - loop (“for” in C/C++; “do” in Fortran)
  - sections
  - single

Loop Construct (work sharing)

Clauses:

- private
- firstprivate
- lastprivate
- reduction
- ordered
- schedule
- nowait

```
#pragma omp parallel shared(n,a,b) private(i)
{ #pragma omp for
      for (i=0; i<n; i++)
            a[i]=i;
   #pragma omp for
      for (i=0; i<n; i++)
            b[i] = 2 * a[i];
}
```

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

Parallel Loop (C/C++)

Style 1

```
#pragma omp parallel
{
 // …
  #pragma omp for
    for (i=0; i<N; i++)
   {
      …
   }// end of for
}// end of parallel
```

Style 2

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    …
}// end of for
```


Parallel Loop (Fortran)

Style 1

```
$!omp parallel
{
 ! ...
  $!omp do
      do i=1,n
        ...
      end do
  $!omp end do
$!omp end parallel
```

Style 2

```
$!omp parallel do
  do i=1,n
     ...
  end do
$!omp end parallel do
```

Loop Scheduling

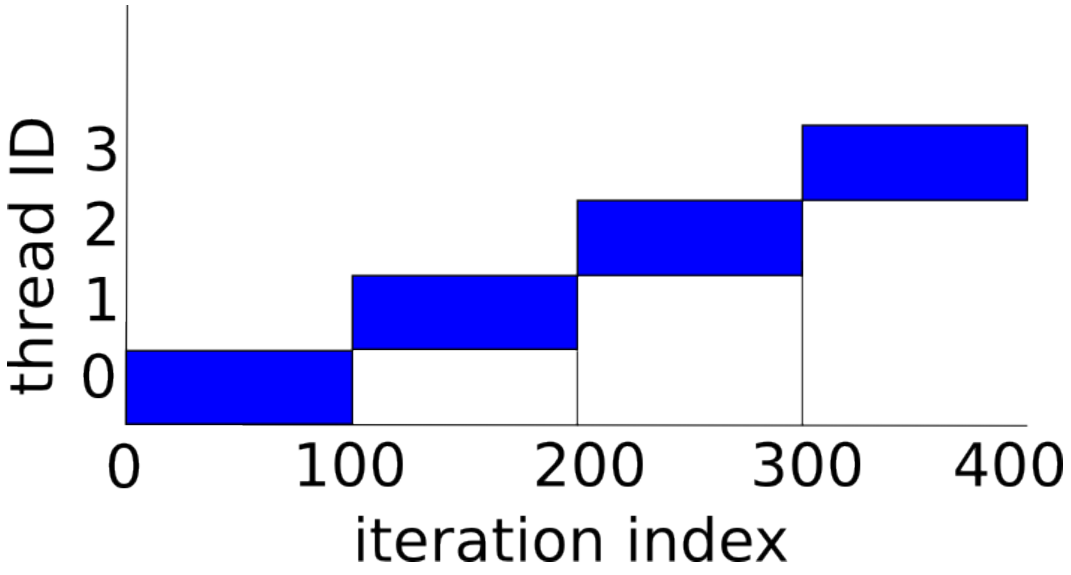
```
#pragma omp parallel for
{
   for (i=0; i<1000; i++)
   { foo(i); }
}
```

How is the loop divided into separate threads?

Scheduling types:

- **static**: each thread is assigned a fixed-size chunk (default)
- **dynamic**: work is assigned as a thread request it
- **guided**: big chunks first and smaller and smaller chunks later
- **runtime**: use environment variable to control scheduling

Static scheduling

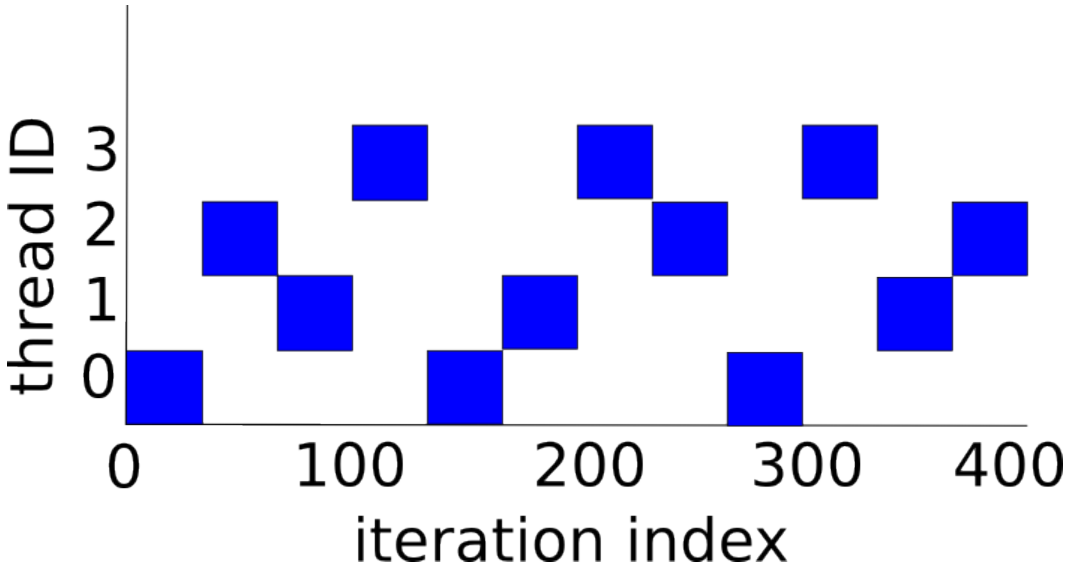


Loop Schedule Example

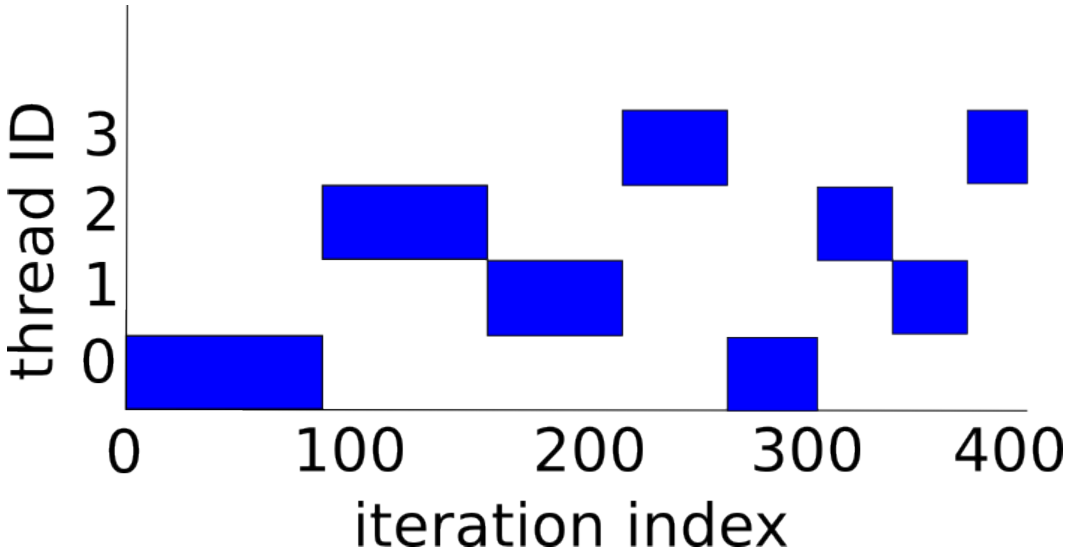
```
#pragma omp parallel for schedule(dynamic,5) \
        shared(n) private(i,j)
   for (i=0; i<n; i++) {
       for (j=0; j<i; j++) {
          foo(i,j);
        } // j-loop
   } // i-loop
} // end of parallel for
```

☞ “dynamic” is useful when the amount of work in foo(i,j) depends on i and j.

Dynamic scheduling



Guided scheduling



Sections

One thread executes one section

- If “too many” sections, some threads execute more than one section (round-robin)
- If “too few” sections, some threads are idle
- We don’t know in advance which thread will execute which section

☞ Each section is executed exactly once

C/C++

```
#pragma omp sections
{
   #pragma omp section
     { foo(); }
   #pragma omp section
      { bar(); }
   #pragma omp section
      { beer(); }
} // end of sections
```

Fortran

```
$!omp sections
   $!omp section
       call foo()
   $!omp end section
   $!omp section
        call bar
   $!omp end section
$!omp end sections
```

Single

A “single” block is executed by one thread

- Useful for initializing shared variables
- We don’t know exactly which thread will execute the block
- Only one thread executes the “single” region; others bypass it.

C/C++

```
#pragma omp single
{
    a = 10;
}
#pragma omp for
{  for (i=0; i<N; i++)
      b[i] = a;
}
```

Fortran

```
$!omp single
    a = 10;
$!omp end single

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


Computing the Sum

- We want to compute the sum of a[0] and a[N-1]:

C/C++

```
sum = 0;
for (i=0; i<N; i++)
  sum += a[i];
```

Fortran

```
sum = 0;
do i=1,n
  sum = sum + a(i)
end do
```

- A “naive” OpenMP implementation (incorrect):

C/C++

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

Race condition!

Fortran

```
sum = 0;
$!omp parallel do
      do i=1,n
            sum = sum + a(i)
      end do
$!omp end parallel do
```

Critical

C/C++

```
#pragma omp critical
{
   //...some stuff
}
```

Fortran

```
$!omp critical
   !...some stuff
$!omp end critical
```

- One thread at a time
  - ALL threads will execute the region eventually
  - Note the difference between “single” and “critical”
- Mutual exclusive

Computing the sum

The correct OpenMP-way:

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sum_local)
  {
    sum_local = 0;
    #pragma omp for
      for (i=0; i<n; i++)
        sum_local += a[i];   // form per-thread local sum

    #pragma omp critical
    {
      sum += sum_local; // form global sum
    }
  }
```

Reduction operation

sum example from previous slide:

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

A cleaner solution:

```
sum = 0;
#pragma omp parallel for \
    shared(...) private(...) \
    reduction(+:sum)
  {
     for (i=0; i<n; i++)
       sum += a[i];
}
```

Reduction operations of +,*,-,& |, ^, &&, || are supported.

Barrier

```
int x = 2;
#pragma omp parallel shared(x)
  {
    int tid = omp_get_thread_num();
    if (tid == 0)
      x = 5;
    else
      printf("[1] thread %2d: x = %d\n",tid,x);

    #pragma omp barrier

    printf("[2] thread %2d: x = %d\n",tid,x);
  }
```

some threads may still have x=2 here

cache flush + thread synchronization

all threads have x=5 here

Resource Query Functions

- Max number of threads

  omp_get_max_threads()

- Number of processors

  omp_get_num_procs()

- Number of threads (inside a parallel region)

  omp_get_num_threads()

- Get thread ID

  omp_get_thread_num()

☞ See OpenMP specification for more functions.

Query function example:

```
#include <omp.h>
int main()
{
  float *array = new float[10000];
  foo(array,10000);
}
```

```
void bar(float *x, int istart, int ipts)
{
  for (int i=0; i<ipts; i++)
    x[istart+i] = 3.14159;
}
```

```
void foo(float *x, int npts)
{
  int tid,ntids,ipts,istart;
#pragma omp parallel private(tid,ntids,ipts,istart)
  {
    tid    = omp_get_thread_num();   // thread ID
    ntids  = omp_get_num_threads();  // total number of threads
    ipts   = npts / ntids;
    istart = tid * ipts;
    if (tid == ntids-1) ipts = npts - istart;
    bar(x,istart,ipts);     // each thread calls bar
  }
}
```

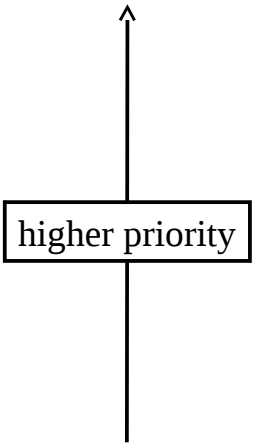
Control the Number of Threads

- Parallel region

  #pragma omp parallel **num_threads**(*integer*)
- Run-time function

  omp_set_num_threads()
- Environment variable

  export OMP_NUM_THREADS=n



☞ High-priority ones override low-priority ones.

OpenMP Environment Variables

- OMP_SCHEDULE
  - Loop scheduling policy
- OMP_NUM_THREADS
  - number of threads
- OMP_STACKSIZE

☞ See OpenMP specification for many others.

Nested Parallel Regions

- Need available hardware resources (e.g. CPUs) to gain performance

```
void main()
{
  #pragma omp parallel
    {
        i = some_index;
        foo(i);
    }
}
```

```
void foo()
{
  #pragma omp parallel
    {
       // some fancy stuff here
    }
}
```

Each thread from main fork a team of threads.

Good Things about OpenMP

- Simplicity
  - In many cases, “the right way” to do it is clean and simple
- Incremental parallelization possible
  - Can incrementally parallelize a sequential code, one block at a time
  - Great for debugging & validation
- Leave thread management to the compiler
- It is directly supported by the compiler
  - No need to install additional libraries (unlike MPI)

Other things about OpenMP

- Data race condition can be hard to detect/debug
  - The code may run correctly with a small number of threads!
  - True for all thread programming, not only OpenMP
  - Some tools may help
- It may take some work to get parallel performance right
  - In some cases, the performance is limited by memory bandwidth (i.e. a hardware issue)

Summary

- Identify compute-intensive, data parallel parts of your code
- Use OpenMP constructs to parallelize your code
  - Spawn threads (parallel regions)
  - In parallel regions, distinguish shared variables from the private ones
  - Assign work to individual threads
    - loop, schedule, etc.
  - Watch out variable initialization before/after parallel region
  - Single thread required? (single/critical)
- Experiment and improve performance