Introduction to OpenMP

Thomas Hauser
Director of Research Computing
University of Colorado Boulder
USGS Alaska Survey
08.26.2014

Research Computing
UNIVERSITY OF COLORADO BOULDER

How to Run Applications Faster?

- There are 3 ways to improve performance:
  - Work Harder
  - Work Smarter
  - Get Help
- Computer Analogy
  - Using faster hardware
  - Optimized algorithms and techniques used to solve computational tasks
  - Multiple computers to solve a particular task

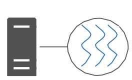
Parallelism on a Processor

- Scaling
  - Using more cores
  - Task parallelism
- Vectorization
  - Wider vectors (same operation on larger number of data)
  - Data parallelism

Parallelism

- Nodes – MPI
  - The following picture shows four stacks aligned side-by-side.

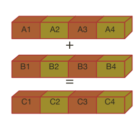
- Threads – OpenMP
  - The following pictures shows …



- Instructions – ILP
  - The following is three lines of instruction code

```
I1: add R1,   R2, R3
I2: sub R4,   R1, R5
I3: xor  R10, R2, R11
```

- Data – SIMD
  - The following picture shows …



OpenMP

- The OpenMP application programming interface (API) supports multi-platform shared-memory parallel programming in
  - C/C++
  - Fortran
- The API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

OpenMP Website

Philosophy

- Goal: Add parallelism to a functioning serial code.
- Requires: Shared memory machine.
- How: Add *compiler directives* to parallelize parts of code.
- Pro: Often very easy to add to existing code.
- Con: Large shared memory machines are expensive.

Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Resources

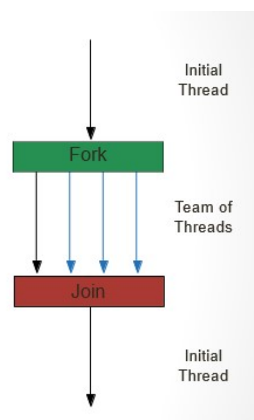
- *Using OpenMP* the book
  - The Mit Press Website
- API Quick Reference
  - OpenMP C & C++ PDF Document
  - OpenMP Fortran PDF Document
- OpenMP v4 full API
  - OpenMP 4.0.0 PDF Document
- OpenMP Examples
  - OpenMP 4.0.2 PDF Document

What's OpenMP Good For?

- C/Fortran + OpenMP sufficient to program shared memory computers
- C/Fortran + MPI + OpenMP a good way to program multicomputers built out of multiprocessors
  - Cluster nodes
  - Intel Phi accelerators

Fork/Join Parallelism

- Program starts as a single thread of execution, initial thread.
- A team of threads is forked at the beginning of a parallel region.
- At the end of a parallel region the threads join (either die or are suspended).
- The following diagram shows the flow between the initial thread through the fork, to the team of threads to join and become the initial thread.



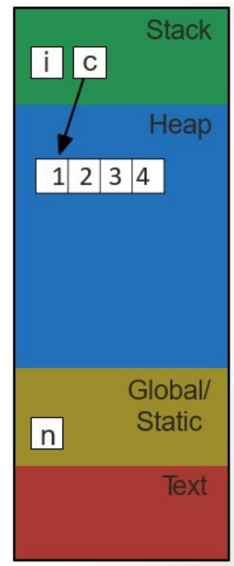
Memory Model

- Contents of memory segments:
  - static variables
  - variables on the run-time stack
  - functions on the run-time stack
  - dynamically allocated data on the heap
  - Below is a snippet of code designed to….

```
#include <stdlib.h> static
const int n = 4: int
main(int argc, char **argv)
{
        int i = 0
        int *c = NULL;
        c = malloc(n * sizeof(int)):
        for (i = 0; I < n; ++i)  {
                c[i] = i +1:
}
free (c):
return(0);

}
```

The following diagram demonstrates Program Memory



Compiler Directive

- Compiler directive in Fortran
- A way for the programmer to communicate with the compiler
- Compiler free to ignore directives
- Syntax:

```
!$OMP <rest of directive>
!$OMP& continuation line
!$ INTEGER :: i !something only in the
  parallel version of the program
```

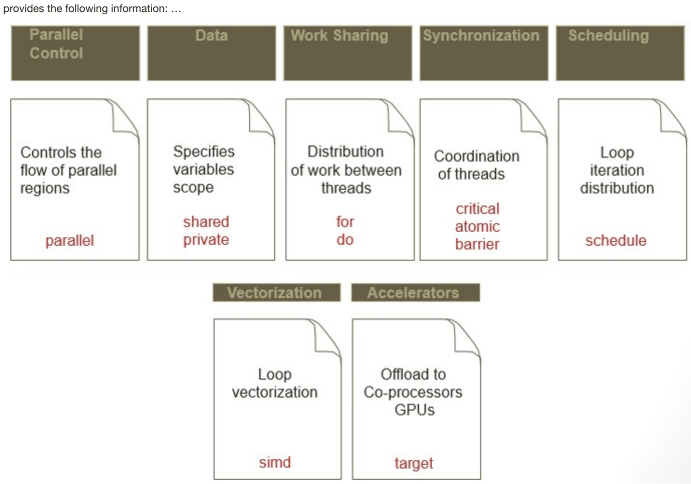
Pragma in C or C++

- Syntax:

```
#pragma omp <rest of directive>
```


Compiler Directives

This table provides the following information: …



Using the GCC compilers with OpenMP

- gfortran -fopenmp –g basic.f90
- gcc -fopenmp -g basic.c
- setenv OMP_NUM_THREADS 12 if tcsh/csh
- export OMP_NUM_THREADS=12 if bash/sh
- ./a.out

Using the Intel compilers with OpenMP

- ifort -fopenmp –g basic.f90
- icc -fpenmp -g basic.c
- setenv OMP_NUM_THREADS 12 if tcsh/csh
- export OMP_NUM_THREADS=12 if bash/sh
- ./a.out

Parallel

Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

Fortran

```
!$OMP parallel do
do i=1,10
   a(i) = b(i) + c(i)
end do
!$OMP end parallel do
```

C

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


OpenMP – Workshare Directives

```
!$OMP PARALLEL [clause]
Tid = omp_get_thread_num()
   !$OMP DO
        Do I = 1, nmax
        End do

   !$OMP END DO
!$OMP END PARALLEL [nowait]
```

parallel directive

- The **`parallel`** directive precedes a block of code that should be executed by *all* of the threads
- Note: execution is replicated among all threads

do directive

- The **`parallel`** directive instructs every thread to execute all of the code inside the block
- If we encounter a **`do`** loop that we want to divide among threads, we use the **`do`** directive

```
!$OMP do
```

Example use of do Directive

The code below is an example of do Directive. The lines beginning with $!OMP are presented in red text.

```
!$omp parallel private(i,j)
Do i = 1, m
   low = a(i)
   high = b(i)
   if (low > high) then
      write(*,*) "Exiting", i
      exit
   endif
End do
!$omp do
do j = low, high
  c(j) = (c(j) - a(i))/b(i)
End do
!$omp end do
!$omp end parallel
```


Parallel Regions

- We tell OpenMP compiler to parallelize code.
- Mark parallel blocks.
- The compiler will spawn threads and split the work up.
- We can tell the compiler the number of threads too.

- Parallel code for C:

```
#pragma omp parallel
{
      …
}
```

- Parallel code for Fortran:

```
!$OMP parallel
      …
!$OMP end parallel
```

OpenMP Also Provides Library Calls.

C function prototypes are in `omp.h.`
Fortran module interface is in `omp_lib`.
For compatibility, you should `#ifdef` guard these calls.
Remember to use the pre-processor for Fortran too (. F90).

- Code for C:

```
#ifdef _OpenMP
#include <omp.h>
#endif
```

- Code for Fortran:

```
#ifdef _OpenMP
    use omp_lib
#endif
```


Library Functions

- OpenMP has functions to find out our thread number and the total number of threads.
  - omp_get_thread_num()
  - omp_get_num_threads()
- We can set the number of threads:
  - Function omp_set_thread_num().
  - Specifying num_threads clause after the parallel directive.
  - With the environment variable **OMP_NUM_THREADS**.

nthreads/nthreads_c.c

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

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

#pragma omp parallel
{
		printf("Hello world! From thread %d\n", omp_get_thread_num());
} /* End omp parallel */

	return(EXIT_SUCCESS);
}
```

There is also a Fortran version nthreads/nthreads_f.f90.

Compiling the program

GCC gcc -fopenmp -o thread_num_c thread_num_c.c
Intel icc -openmp -o thread_num_c thread_num_c.c
IBM xlc -qsmp=omp -o thread_num_c thread_num_c.c

- Execute the program, specifying different numbers of threads.
  1. ./thread_num_c
  2. env OMP_NUM_THREADS=1 ./thread_num_c
  3. env OMP_NUM_THREADS=24 ./thread_num_c
- What is the output?
  - Threads printed out their identification number.
  - Random order of numbers. Threads execute independently and in general order will be random.

Data Scoping – shared vs private

- Default: All data in a parallel region is shared
- This includes global data (global/static variables, C++ class variables)
- Exceptions:
  - Local data within enclosed function calls are private (Note: Inlining must be treated correctly by compiler!) unless declared static)
  - Loop variables of parallel ("sliced") loops are private (cf. workshare constructs)
- Due to stack size limits it may be necessary to make large arrays static
  - This presupposes it is safe to do so!
  - If not: make data dynamically allocated
  - As of OpenMP 3.0: OMP_STACKSIZE may be set at run time (increase thread-specific stack size):

    $ setenv OMP_STACKSIZE 100M

Private Variables – Masking

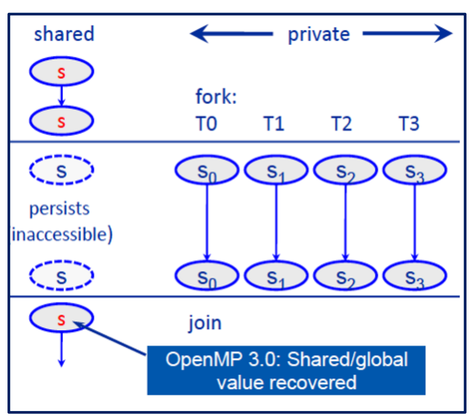
The following diagrams…

```
real :: s

        s = …
!$omp parallel private(s)

        s = …
        ;;; = … + s

!$omp end parallel
        … = … + s
```



Declaring Private Variables

The following code is …

```
do j = 1,nj
  do i = 1, ni
    a(i,j) = min(a(i,j), a(i,j)+tmp)
  end do
end do
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable **i**

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

private (***<variable list>*** **)**

Example Use of private Clause

```
!$OMP parallel do private(i)
DO j = 1, nj
   DO i = 1, ni
      a(i,j) = MIN(a(i,j),a(i,k)+tmp)
   END DO
END DO

#pragma omp for private(i)
for (j=0;j<nj;j++)
  for (i=0; i<ni; i++) {
    a[j,i] = a[j,i]+tmp
}
```


firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

Scoping – firstprivate

The following diagrams show…

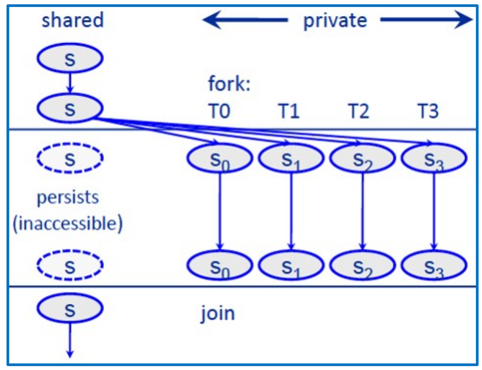
```
real :: s

        s = …
!$omp parallel &
!$omp firstprivate(s)

        … = … + s
        call foo()

!$omp end parallel
        … = … +  s
```

If foo() references or defines s (e.g. by host association), it may work on a copy of s.



Use of firstprivate

```
program wrong
I = 10
!$OMP PARALLEL
   PRIVATE(I)
I= I + 1
print *, I
!$OMP END PARALLEL
```

```
program correct
I = 10
!$OMP PARALLEL
   FIRSTPRIVATE(I)
I= I + 1
print *, I
!$OMP END PARALLEL
```

lastprivate Clause

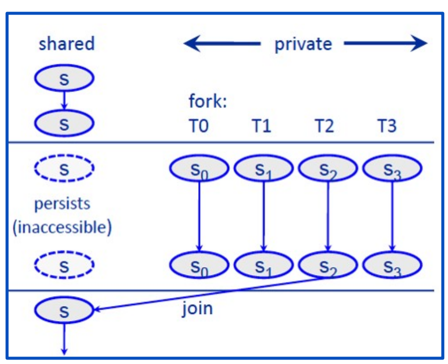
- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- **`lastprivate`** clause: used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration

Scoping – lastprivate

The following diagram shows….

```
real :: s

    s = …
!$omp parallel &
!$omp lastprivate(s)
!$omp do
    do i = …
        s = …
    end do
!$omp end do
!$omp end parallel
    … = … + s
```



Function omp_get_num_procs

- Returns number of physical processors available for use by the parallel program
- C: **int omp_get_num_procs**(void);
- Fortran:

```
interface
   function omp_get_num_procs ()
      use omp_lib_kinds
      integer ( kind=omp_integer_kind ) :: omp_get_num_procs
   end function omp_get_num_procs
end interface
```


Function omp_set_num_threads

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```
subroutine omp_set_num_threads (t)
integer, intent(in) :: t
```

void omp_set_num_threads(int *num_threads*);

Variables

- We must tell the compiler how to use variables.
  - A `shared` variable has the same address in memory in every thread.
  - A `private` variable has a different address in memory in every thread.
  - A `firstprivate` is private, however it is pre-initialize.
- Clauses specifies the scope of variables.

- Variables in C

```
#pragma omp parallel    \
        default(none)   \
        shared(x)       \
        private(i,j)    \
```

- Variables in Fortran

```
!$OMP parallel          &
!$OMP default(none)     &
!$OMP shared(x)         &
!$OMP private(i,j)
     ...
!$OMP end parallel
```


The default is shared.

- Try and always specify `default(none)`, so as not to confuse a variables behavior. Then explicitly define every variable.
- ▸ In C, you are able to declare variables within structured blocks to reduce it's scope. This will make it a private variable. For example, `i` is shared while `j` is private.

Default in C

```
int i = 0;
 #pragma omp parallel \
         default(none)   \
         shared(i)
{
              int j = 0;
 }
```

Questions?

Online Survey

Thomas Hauser's Email

License