Introduction to OpenMP

- *OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.*

Introduction to OpenMP

- *OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.*
- OpenMP is a specification that compilers can implement. The latest version is 4.0. GCC implements version 3.1. VisualStudio C/C++ implements version 2.0.

Introduction to OpenMP

- *OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.*
- OpenMP is a specification that compilers can implement. The latest version is 4.0. GCC implements version 3.1. VisualStudio C/C++ implements version 2.0.
- Relatively easy way to get moderate parallelism on shared-memory machines.

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
- OpenMP combined with C, C++ or Fortran creates a multithreaded program where the threads share the address space.

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
- OpenMP combined with C, C++ or Fortran creates a multithreaded program where the threads share the address space.
- The goal of OpenMP was to make it easier for programmers to convert single-threaded code to multithreaded. The two key concepts are:

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
- OpenMP combined with C, C++ or Fortran creates a multithreaded program where the threads share the address space.
- The goal of OpenMP was to make it easier for programmers to convert single-threaded code to multithreaded. The two key concepts are:
  - sequential equivalence: yields the same results whether it executes using one thread or many threads.

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
- OpenMP combined with C, C++ or Fortran creates a multithreaded program where the threads share the address space.
- The goal of OpenMP was to make it easier for programmers to convert single-threaded code to multithreaded. The two key concepts are:
  - sequential equivalence: yields the same results whether it executes using one thread or many threads.
  - incremental parallelism: a style of parallel programming where a program evolves incrementally from a sequential program to a parallel program.

What is OpenMP?

- OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
- OpenMP combined with C, C++ or Fortran creates a multithreaded program where the threads share the address space.
- The goal of OpenMP was to make it easier for programmers to convert single-threaded code to multithreaded. The two key concepts are:
  - sequential equivalence: yields the same results whether it executes using one thread or many threads.
  - incremental parallelism: a style of parallel programming where a program evolves incrementally from a sequential program to a parallel program.
  - OpenMP is an explicit parallel programming approach so the compiler doesn't guess how to exploit concurrency.

OpenMP Basics

- To create threads, the programmer designates blocks of code that are to be run in parallel with the pragma

```
#pragma omp parallel
```

OpenMP Basics

- To create threads, the programmer designates blocks of code that are to be run in parallel with the pragma
  `#pragma omp parallel`
  (See Wikipedia page on pragmas)
- The environment variable `OMP_NUM_THREADS` determines the number of threads used at run time.

OpenMP Basics

- To create threads, the programmer designates blocks of code that are to be run in parallel with the pragma
  `#pragma omp parallel`
  (See Wikipedia page on pragmas)
- The environment variable `OMP_NUM_THREADS` determines the number of threads used at run time.
- Here is an example program that uses the pragma.

```
/* OpenMP/hello_1/hello_1.c */
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv)
{
    #pragma omp parallel
    {
        printf("hello world\n");
    }
}
```

- Compile with the `-fopenmp` flag for the GCC compiler

Shared/private variables

- A variable allocated prior to a parallel region is shared between the threads (in most cases).

Shared/private variables

- A variable allocated prior to a parallel region is shared between the threads (in most cases).
- If a variable is declared inside a parallel region, then it is *private* or *local* to a thread.

Shared/private variables

- A variable allocated prior to a parallel region is shared between the threads (in most cases).
- If a variable is declared inside a parallel region, then it is *private* or *local* to a thread.
- The number of threads can also be specified by the programmer via an OpenMP runtime library call omp_set_num_threads(). Or it can also be set by the num_threads() clause as shown below.

```
#pragma omp parallel num_threads(4)
```

hello_2 example

```
/* lab/OpenMP/hello_2/hello_2.c */
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv)
{
    int i=100; //this becomes a shared variable
    omp_set_num_threads(4);//OpenMP library call
    #pragma omp parallel
    {
        int id; //private to each thread
        id = omp_get_thread_num(); //OpenMP library call
        printf("i = %d on thread %d\n", i, id);
    }
}
```

OpenMP pragmas

An OpenMP construct is defined to be a directive(pragma) plus a block of code with the following conditions:

OpenMP pragmas

An OpenMP construct is defined to be a directive(pragma) plus a block of code with the following conditions:

- The block of code must be structured: one point of entry at the top and a single point of exit at the bottom.

OpenMP pragmas

An OpenMP construct is defined to be a directive(pragma) plus a block of code with the following conditions:

- The block of code must be structured: one point of entry at the top and a single point of exit at the bottom.
- No branching into or out of the structured block is allowed.

OpenMP pragmas

An OpenMP construct is defined to be a directive(pragma) plus a block of code with the following conditions:

- The block of code must be structured: one point of entry at the top and a single point of exit at the bottom.
- No branching into or out of the structured block is allowed.
- No `return` statement is allowed inside the structured block.

OpenMP pragmas

An OpenMP construct is defined to be a directive(pragma) plus a block of code with the following conditions:

- The block of code must be structured: one point of entry at the top and a single point of exit at the bottom.
- No branching into or out of the structured block is allowed.
- No `return` statement is allowed inside the structured block.
- The only branching statement allowed is one that ends the program (exit()).

```
#pragma omp <directive-name> [<clause> [<clause>] ...]
```

Work-Sharing

Different code to map to different threads. Common examples are loop splitting and separate sections.

Work-Sharing

Different code to map to different threads. Common examples are loop splitting and separate sections.
Loop splitting:

- The programmer identifies the most time-consuming loops in their program.
- The loops are then restructured, if necessary, so that the loop iterations are largely independent.
- The program is then parallelized by mapping different groups of loop iterations into different threads.

Work-Sharing

Different code to map to different threads. Common examples are loop splitting and separate sections.

Loop splitting:

- The programmer identifies the most time-consuming loops in their program.
- The loops are then restructured, if necessary, so that the loop iterations are largely independent.
- The program is then parallelized by mapping different groups of loop iterations into different threads.

Sections

- The programmer identifies separate sections of code that can be done by separate threads.

Pragmas for Work-Sharing

- for directive:

```
#pragma omp for [<clause> [<clause>] ... ]
<for-loop statement>
```

Pragmas for Work-Sharing

- for directive:

```
#pragma omp for [<clause> [<clause>] ... ]
<for-loop statement>
```

- section directive:

```
#pragma omp sections [<clause> [<clause>] ... ]
{
    [#pragma omp section]
    <C/C++ structured block executed by processor i>
    [#pragma omp section]
    <C/C++ structured block executed by processor j>
...
}
```

Pragmas for Work-Sharing

- for directive:

```
#pragma omp for [<clause> [<clause>] ... ]
<for-loop statement>
```

- section directive:

```
#pragma omp sections [<clause> [<clause>] ... ]
{
    [#pragma omp section]
    <C/C++ structured block executed by processor i>
    [#pragma omp section]
    <C/C++ structured block executed by processor j>
...
}
```

- single directive: The first thread that encounters the block of code executes it, other threads skip the block and wait at the end of the single construct.

```
#pragma omp single
{<structured block>}
```

How Work-Sharing Pragmas Work

- There is an implicit barrier at the end of any work sharing pragma. This can be removed by appending the `nowait` clause at the end of the `for` pragma.

How Work-Sharing Pragmas Work

- There is an implicit barrier at the end of any work sharing pragma. This can be removed by appending the `nowait` clause at the end of the `for` pragma.
- The loop index is automatically made private for each thread.

How Work-Sharing Pragmas Work

- There is an implicit barrier at the end of any work sharing pragma. This can be removed by appending the `nowait` clause at the end of the `for` pragma.
- The loop index is automatically made private for each thread.
- The loop iterations are divided among the threads in an order decided by the system. This, however, can be controlled by the programmer.

How Work-Sharing Pragmas Work

- There is an implicit barrier at the end of any work sharing pragma. This can be removed by appending the `nowait` clause at the end of the `for` pragma.
- The loop index is automatically made private for each thread.
- The loop iterations are divided among the threads in an order decided by the system. This, however, can be controlled by the programmer.
- It is common to have parallel construct followed by a for construct. We can combine the two as follows:
  `#pragma omp parallel for`

Work Sharing Example

- Sequential code. Assume that the combine function doesn't take long and must be called sequentially.

```
double result;
double answer = 0.0;
for (i=0; i<N; i++) {
    result = bigcomputation(i);
    combine(answer, result);
}
```

Work Sharing Example

- Sequential code. Assume that the combine function doesn't take long and must be called sequentially.

```
double result;
double answer = 0.0;
for (i=0; i<N; i++) {
    result = bigcomputation(i);
    combine(answer, result);
}
```

- The parallelized version. Notice how we divided the loop into two independent loops.

```
double result[N];
double answer = 0.0;
#pragma omp parallel for
for (i=0; i<N; i++)
    result[i] = bigcomputation(i);

for (i=0; i<N; i++)
    combine(answer, result);
```

Data Environment Clauses

- The private(<variable-list>) clause directs the compiler to create a private or local variable for each name included in the list. The names in the list must have been defined and bound to shared variables prior to the the parallel region. The initial values of these new private variables are undefined. Furthermore, the values of these variables after the parallel region are undefined as well.

Data Environment Clauses

- The private(<variable-list>) clause directs the compiler to create a private or local variable for each name included in the list. The names in the list must have been defined and bound to shared variables prior to the the parallel region. The initial values of these new private variables are undefined. Furthermore, the values of these variables after the parallel region are undefined as well.
- The reduction(<operator>:<variables>) clause combines a set of values into a single value using the specified binary, associative operator. The variables are initialized with the identity value, and then each thread gets a private copy to work on. At the end of the parallel region, the values are combined and assigned to the respective variable after the parallel region. Operators (in C/C++): +, /, *, -, &, |, ∧, &&, ||.

Data Environment Clauses

- The private(<variable-list>) clause directs the compiler to create a private or local variable for each name included in the list. The names in the list must have been defined and bound to shared variables prior to the the parallel region. The initial values of these new private variables are undefined. Furthermore, the values of these variables after the parallel region are undefined as well.
- The reduction(<operator>:<variables>) clause combines a set of values into a single value using the specified binary, associative operator. The variables are initialized with the identity value, and then each thread gets a private copy to work on. At the end of the parallel region, the values are combined and assigned to the respective variable after the parallel region. Operators (in C/C++): +, /, *, -, &, |, ∧, &&, ||.
- Other clauses: firstprivate(<variable-list>) and lastprivate(<variable-list>). The firstprivate clause takes the value prior to the parallel region as the initial value in the region. The lastprivate clause takes the value assigned in the last loop iteration and makes that the value of the variable after the parallel region.

Work Sharing Example - OpenMP Sum

```
/* lab/OpenMP/sum/openmp_sum.c */
/* appropriate header files */
int main(int argc, char **argv)
{
    int i;
    int n;
    int *array;
    long int sum = 0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <n>\n", argv[0]);
        exit(1);
    }
    n = atoi(argv[1]);
    printf("Number of elements to add = %d\n", n);
    array = (int *) malloc(sizeof(int)*n);

    #pragma omp parallel for
    for (i=0; i<n; i++)
        array[i] = 1;

    #pragma omp parallel for reduction(+:sum)
    for (i=0; i<n; i++) {
        sum +=  array[i];
    }
    printf(" sum =  %ld\n", sum);
    exit(0);
}
```

Work Sharing Example - OpenMP Pi

```
/* lab/OpenMP/pi/pi.c */
/* appropriate header files */
int main(int argc, char **argv)
{
    int i;
    int num_steps = 100000000;
    double x, pi, step, sum = 0.0;
    if (argc > 1) {
        num_steps = atoi(argv[1]);
    }
    step = 1.0/(double) num_steps;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf(" pi =  %.9lf\n", pi);
    exit(0);
}
```

Function calls in loops

- In general, the compiler will not parallelize a loop that involves a function call unless it can guarantee that there are no dependencies between iterations.

Function calls in loops

- In general, the compiler will not parallelize a loop that involves a function call unless it can guarantee that there are no dependencies between iterations.
- A good strategy is to inline function calls within loops. If the compiler can inline the function, it can usually verify lack of dependencies.

Function calls in loops

- In general, the compiler will not parallelize a loop that involves a function call unless it can guarantee that there are no dependencies between iterations.
- A good strategy is to inline function calls within loops. If the compiler can inline the function, it can usually verify lack of dependencies.
- See example `OpenMP/parallel-for/ex1.c` for one that does parallelize. See example `OpenMP/parallel-for/ex2.c` for one that doesn't parallelize. Relevant code snippets shown below.

```
#pragma omp parallel for
   for (i = 0; i < n; i++)
       a[i] = sqrt(2 * i * 3.14159) ;

#pragma omp parallel for
   for (i = 0; i < n; i++)
       a[i] = sqrt(2 * i * 3.14159) * random() ;
```

OpenMP Runtime Library

- omp_set_num_threads(int)
- omp_get_num_threads()
- omp_get_thread_num()
- Lock functions for explicit manipulation of locks.

```
/* lab/OpenMP/hello_3/hello_3.c */
/* appropriate */
int main(int argc, char **argv) {
    int id, numThreads;
    #pragma omp parallel private (id, numThreads)
    {
        id = omp_get_thread_num(); //OpenMP library call
        numThreads = omp_get_num_threads();
        printf("I am thread %d out of  %d threads\n", id, numThreads);
    }
}
```

Synchronization Constructs

- `flush` defines a synchronization point where memory consistency is enforced.
  #pragma omp flush [(<variable-list>)]
  If the variable list is omitted, then flush operates on all variables visible to the calling thread.

Synchronization Constructs

- `flush` defines a synchronization point where memory consistency is enforced.
  #pragma omp flush [(<variable-list>)]
  If the variable list is omitted, then flush operates on all variables visible to the calling thread.
- critical implements a critical section for mutual exclusion. An optional name can be provided to support disjoint sets of critical sections.
  #pragma omp critical [(<variable-name>)]
  `<structured block>`

Synchronization Constructs

- `flush` defines a synchronization point where memory consistency is enforced.
  #pragma omp flush [(<variable-list>)]
  If the variable list is omitted, then flush operates on all variables visible to the calling thread.
- critical implements a critical section for mutual exclusion. An optional name can be provided to support disjoint sets of critical sections.
  #pragma omp critical [(<variable-name>)]
  `<structured block>`
- barrier provides a synchronization point at which the threads wait until every member of the team has arrived before any threads continue.
  #pragma omp barrier

Critical Section: Example 1

```
/* lab/OpenMP/hello_4/hello_4.c */
/* appropriate header files */
int main(int argc, char **argv)
{
    int i=100; //this becomes a shared variable
    #pragma omp parallel
    {
        #pragma omp critical
        { i++; }
    }
    printf("i = %d after parallel section\n", i);
}
```

Critical Section: Example 2

```
double result;
double answer = 0.0;
#pragma omp parallel for private(result)
for (i=0; i<N; i++) {
    result = bigcomputation(i);
    #pragma omp critical
    combine(answer, result);
}
```

Assumes that calls to combine can be made in any order.

Low Level Synchronization

- `void omp_init_lock(omp_lock_t *lock)`
- `void omp_destroy_lock(omp_lock_t *lock)`
- `void omp_set_lock(omp_lock_t *lock)`
- `void omp_unset_lock(omp_lock_t *lock)`
- `int omp_test_lock(omp_lock_t *lock)`

The lock functions guarantee that the lock variable is consistently updated between threads, but do not imply a flush of other variables. Therefore, programmers using locks must call `flush` explicitly.

Load Balancing in Loops

Load balancing is achieved via the schedule clause to the `for` construct.
The syntax is: schedule (<sched> [, <chunk>])

- schedule (static [, chunk]): Static load balancing by iterations.

Load Balancing in Loops

Load balancing is achieved via the schedule clause to the `for` construct.
The syntax is: schedule (<sched> [, <chunk>])

- schedule (static [, chunk]): Static load balancing by iterations.
- schedule (dynamic [, chunk]): Dynamic load balancing. Similar to a centralized workpool queue.

Load Balancing in Loops

Load balancing is achieved via the schedule clause to the `for` construct.
The syntax is: schedule (<sched> [, <chunk>])

- schedule (static [, chunk]): Static load balancing by iterations.
- schedule (dynamic [, chunk]): Dynamic load balancing. Similar to a centralized workpool queue.
- schedule (guided [, chunk]): Dynamic load balancing with decreased number of scheduling decisions. Number of iterations assigned decrease exponentially down to chunk size.

Load Balancing in Loops

Load balancing is achieved via the schedule clause to the `for` construct.
The syntax is: schedule (<sched> [, <chunk>])

- schedule (static [, chunk]): Static load balancing by iterations.
- schedule (dynamic [, chunk]): Dynamic load balancing. Similar to a centralized workpool queue.
- schedule (guided [, chunk]): Dynamic load balancing with decreased number of scheduling decisions. Number of iterations assigned decrease exponentially down to chunk size.
- schedule (runtime): Get scheduling at runtime from environment variable `OMP_SCHEDULE`

See example `OpenMP/load-balancing/lb_demo.c`.

Schedule Clause: Example

```
#define N 1000
double result;
double answer = 0.0;
#pragma omp parallel for private(result) schedule(dynamic, 10)
for (i=0; i<N; i++) {
    result = bigcomputation(i);
    #pragma omp critical
    combine(answer, result);
}
```

Case Study: Wave Simulation

- Initially added a parallel and `for` pragmas around the inner doubly-nested loop as it updates the matrix in parallel. However that gave poor performance (slower than serial!) because of contention for variables.
- Then added private clauses for relevant variables. That finally gave a speedup of about 1.6 with 4 threads on the ghost row based wave program. This was on a system with a four core processor.
- Also experimented with schedule but got no further improvement.
- The pragma used was:

```
#pragma omp parallel for private (i, j, iS, iN, jE, jW)
```

References

- *OpenMP: Simple, portable, scalable SMP programming*. http://www.openmp.org
- *Parallel Programming in OpenMP* by Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald and Ramesh Menon. Morgan Kaufmann Publishers, 2000.
- Appendix A in *Patterns for Parallel Programming* by Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill. Addison Wesley, 2005.
- *Reap the Benefits of Multithreading without All the Work* by Kang Su Gatlin and Pete Isensee, MSDN Magazine, October 2005.