Lecture 11

Topics

- Introduction
- OpenMP
  - Compiler directives
  - Some Examples
  - Library functions
  - Environment variables

Introduction

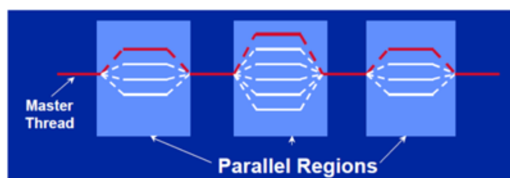
- OpenMP is:
  - a standard for parallel programming in C, C++, and Fortran on shared memory computers.
  - portable, just like MPI.
  - widely supported across vendors, including Intel, Microsoft, HP, SGI, IBM, Sun, etc…
  - gaining importance due to the availability of **multi-core** CPUs.
- GCC 4.2, Intel Compiler 7.x, and VS2005 starts supporting OpenMP 2.0+. (VS2010 and likely VS11: OpenMP 2.0 … Orz)
- OpenMP is now 3.1 (announced on 2011.07, gcc 4.7, Intel compiler 12)
- OpenMP consists of:
  - Compiler directives (comments)
  - Library functions (function calls, API)
  - Environment variables

Shared Memory Parallelization

- Multithread programming
  - Thread: An execution entity having a serial flow of control and an associated stack.
- All processors can access all the memory in the parallel system (one address space).
- The time to access the memory may not be equal for all processors. (NUMA)
- Parallelizing on a SMP does not reduce CPU time.
  - It reduces wall-clock time.
- Parallel execution is achieved by generating multiple threads which execute in parallel.
- Number of threads (in principle) is independent of the number of processors

OpenMP Execution Model
Fork-Join model

- Master (or initial) thread spawns a team of threads as needed.
- Parallelism (team: master + workers) is added incrementally: i.e. the sequential program evolves into a parallel program.



Tim Mattson and Rudolf Eigenmann (1999), “OpenMP: An API for Writing Portable SMP Application Software”

Example

```
for(i=0;i<n;i++) {
    x[i]=2.0*x[i]+3.0*sqrt(x[i])-exp(x[i]);
}
```

```
#pragma omp parallel for
for(i=0;i<n;i++) {
    x[i]=2.0*x[i]+3.0*sqrt(x[i])-exp(x[i]);
}
```

Easy to create multithreads, and each thread can be scheduled by OS to different processors to take advantage of multiple CPU cores.

Syntax

- OpenMP consists of:
  - Compiler directives
  - *Library functions (function calls, API)*
  - *Environment variables*

#pragma omp *directive* [*clause* [*clause*] ...]

7

OpenMP

Compiler directives
Library functions
Environment variables

8

Compiler directives

Conditional compilation
Parallel construct
Work-sharing constructs
Work-tasking constructs
Synchronization

9

Compiler directives
conditional compilation

- A macro "_OPENMP" is defined with the value yyyymm
  - yyyy: year of the implemented OpenMP
  - mm: month of the implemented OpenMP

```
00.cpp
#include <iostream>

using namespace std;

int main() {
  #ifdef _OPENMP
    cout << "OpenMP: " << _OPENMP << endl;
  #else
    cout << "Your compiler does not support OpenMP." << endl;
  #endif
  return 0;
}
```

```
icpc -openmp 00.cpp
OpenMP: 200805
```

```
icpc 00.cpp
Your compiler does not support OpenMP.
```

10

Compiler directives

Conditional compilation
Parallel construct
Work-sharing constructs
Work-tasking constructs
Synchronization

11

Compiler directives
parallel construct

- A parallel construct defines the structured block followed being executed by a team of threads.

```
01a.cpp
#include <iostream>

using namespace std;

int main() {
  #pragma omp parallel
  {
    cout << "Hello World" << endl;
  }
  return 0;
}
```

A construct

```
./01a.exe
Hello World
Hello World
```

Structured block: an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom.

12

Examples: structured block

```
cout <<"Hello World" << endl;
```

```
{
  a=a+b;
  c++;
  cout << "Hello World" << endl;
}
```

```
{
    cout << "Hello World" << endl;
    cout << "Hello Kitty" << endl;
    cout << "Hello Penguin" << endl;
}
```

```
for(i=0;i<100;i++) {
    cout << "Hello World" << endl;
    cout << "Hello Kitty" << endl;
    cout << "Hello Penguin" << endl;
}
```

13

Examples: non-structured block

```
{
  a=a+b;
  goto abc
  c++;
  cout << "Hello World" << endl;
}
:abc
```

```
for(i=0;i<100;i++) {
    cout << "Hello World" << endl;
    if(i==20) break;
    cout << "Hello Penguin" << endl;
}
```

14

Compiler directives
parallel construct

#pragma omp parallel *[clause [[,] clause] …] new-line*
structured-block

Clauses

num_threads(integer-expression)

private(list)
default(shared | none)
shared(list)
firstprivate(list)
**copyin (operator:list)*

reduction(operator: list)

if(scalar-expression)

OpenMP Application Program Interface
Section 2.9 Data Environment

15

Compiler directives
parallel construct

- num_threads(n): defines the number of threads to be created.
  - n: must be an integer expression (numbers, variables, …)

```
02a.cpp
#include <iostream>

using namespace std;

int main() {
  #pragma omp parallel num_threads(8)
  {
    cout << "Hello World" << endl;
  }
  return 0;
}
```

```
./01a.exe
Hello WorldHello World

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

16

Compiler directives
parallel construct

- Data sharing: defines the sharing attribute of variables between threads in the parallel region
  - private(list): list the variables that are private in each thread
  - shared(list): list the variables that are shared between threads
  - default(shared|none): set the default sharing attribute of variables not listed.
- Private variables
  - Private variables are undefined on entry and exit of the parallel region.
  - The value of the original variable (before the parallel region) is **UNDEFINED** after the parallel region.
  - Variables declared in the parallel region are private variables.
  - parallel for 's index variables are private variables.
- Shared variables
  - These variables are shared between all threads
  - They cannot be declared in the structure block followed.

17

```
03a.cpp
#include <iostream>
using namespace std;
int main() {
  int count=17;
  #pragma omp parallel private(count)
  {
    count=0;
    for(int i=0;i<10;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

```
OMP_NUM_THREADS=8 ./03a.exe
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
My count: 10
Main count: 17
```

The variable count inside the parallel construct is in fact a new variable that has nothing to do with the count declared in the main().!

18

04a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: My count:
10919331091933
My count: 2170209

My count: 3213347
My count: 3751776
My count: 4269572
My count: 4272164
My count: 4347902
Main count: 4347902
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1501215
My count: 2501215
My count: 3501215
My count: 4501215
My count: 5501215
My count: 6501215
My count: 7501215
My count: 1000017
Main count: 1000017
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1089747
My count: 2126479
My count: 2240935
My count: 3165806
My count: 3345987
My count: 3364074
My count: 4210268
My count: 4425132
Main count: 4425132
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1152417
My count: 2249624
My count: 2274868
My count: 3346454
My count: 3858762
My count: 4402852
My count: 4406426
My count: 4465075
Main count: 4465075
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1000017
My count: 2000017
My count: 3000017
My count: 4000017
My count: 5000017
My count: 6000017
My count: 7000017
My count: 8000017
Main count: 8000017
```

The variable count is now the one declared in the main(), and are shared by all threads. The show results are known as race condition.

19

05a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel firstprivate(count)
  {
    for(int i=0;i<10;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727

My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727
My count: 27
My count: 27
My count: 27

My count: 27
My count: 27
My count: 27
Main count: 17
```

```
OMP_NUM_THREADS=8 ./05a.exe
My count: My count: 2727
My count: 27
My count: 27
My count: 27

My count: 27
My count: 27
My count: 27
Main count: 17
```

Firstprivate "first" initializes newly created private variables within each thread with the original variable. (If the variable is an object, it is created using copy constructor).

20

06a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel default(none)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
icpc -O2 -Wall -openmp 06a.cpp -o 06a.exe stopWatch.o
06a.cpp(6): error: "count" must be specified in a variable list at
enclosing OpenMP parallel pragma
        #pragma omp parallel default(none)
        ^
06a.cpp(6): error: "std::cout" must be specified in a variable list at
enclosing OpenMP parallel pragma
        #pragma omp parallel default(none)
        ^
compilation aborted for 06a.cpp (code 2)
make: *** [06a.exe] Error 2
```

21

07a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int sum=0;
  #pragma omp parallel reduction(+:sum)
  {
    for(int i=0;i<=10;i++) {
      sum+=i;
    }
    cout << "My sum: " << sum << endl;
  }
  cout << "Main sum: " << sum << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./07a.exe
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 440
```

```
OMP_NUM_THREADS=4 ./07a.exe
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 220
```

```
OMP_NUM_THREADS=2 ./07a.exe
My sum: 55
My sum: 55
Main sum: 110
```

22

08a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int sum=0, n;

  cout << "\nEnter 0 to run single thread:";
  cin >> n;
  #pragma omp parallel reduction(+:sum) if(n)
  {
    for(int i=0;i<=10;i++) {
      sum+=i;
    }
    cout << "My sum: " << sum << endl;
  }
  cout << "Main sum: " << sum << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./08a.exe

Enter 0 to run single thread:1
My sum: 55
My sum: 55
My sum: 55
My sum: 55
Main sum: 220
```

```
OMP_NUM_THREADS=4 ./08a.exe

Enter 0 to run single thread:0
My sum: 55
Main sum: 55
```

23

Reduction operators

	Operator	Initialization value	Remark
Arithmetic Operations	+	0	
	*	1	
	-	0	
Bitwise Operations	&	~0	
	|	0	
	^	0	
Logical Operations	&&	1	
	||	0	
Extremes	max	n/a	New in 3.1
	min	n/a	New in 3.1

24

Compiler directives
parallel construct

#pragma omp parallel *[clause [[,] clause] …] new-line*
structured-block

Clauses

- *num_threads(integer-expression)*
- *private(list)*
- *default(shared|none)*
- *shared(list)*
- *firstprivate(list)*
- **copyin(list) – not discussed …*
- *reduction(operator: list)*
- *if(scalar-expression)*

OpenMP Application Program Interface
Section 2.9 Data Environment

Parallel construct create multiple threads that runs the same code simultaneously!

25

Summary

- So far, we have introduced how to create multiple threads, and define their data environment (sharing variables, …)
- But, we haven't said anything about how to share work load between different threads

26

Compiler directives

Conditional compilation
Parallel construct
Work-sharing constructs
Work-tasking constructs
Synchronization

27

Compiler directives
Work-sharing constructs

- Parallel construct creates a team of threads that runs the code in the parallel region independently.
- Working-sharing constructs are bind/enclosed in an active parallel region to share work by the team of threads.
  - for construct – share workload in for-loops
  - sections construct – partition workloads into sections
  - single construct – allowing one thread to work

28

Compiler directives
for construct

#pragma omp for *[clause [[,] clause] …] new-line*
for-loop

Clauses

- *private(list)*
- *firstprivate(list)*
- *lastprivate(list)*
- *reduction(operator: list)*
- *schedule(kind[, chunk_size])*
- **collapse(n)*
- *ordered*
- *nowait*

OpenMP Application Program Interface
Section 2.5 Worksharing constructs

schedule(static, *chunk_size*)
schedule(dynamic, *chunk_size*)
schedule(guided, *chunk_size*)
schedule(runtime)

29

11a.cpp
```
#include <iostream>
#include <omp.h>

using namespace std;

int main(int argc, char **argv) {
  long i, sum=0;
  double t1;

  t1 = omp_get_wtime();
  for(i=1;i<1000000000;i++) {
    sum += i;
  }
  t1 = omp_get_wtime() - t1;

  cout << sum << " (" << t1 << ")" << endl;
  return 0;
}
```

Recall: MPI_Wtime()

```
./11a.exe

499999999500000000 (1.17483)
```

30

```
12a.cpp
#include <iostream>
#include <omp.h>

using namespace std;

int main(int argc, char **argv) {
  long i, sum=0;
  double t1;

  t1 = omp_get_wtime();
  #pragma omp parallel
  {
    #pragma omp for reduction(+:sum)
    for(i=1;i<1000000000;i++) {
      sum += i;
    }
  }
  t1 = omp_get_wtime() - t1;

  cout << sum << " (" << t1 << ")" << endl;
  return 0;
}
```

variable *i* is implicitly private.

Recall: MPI_Reduce()

```
./11a.exe
499999999500000000 (1.17483)
```

```
OMP_NUM_THREADS=2 ./12a.exe
499999999500000000 (0.602516)
```


Compiler directives
for *construct*, schedule *clause*

- *static [, chunk]*
  - *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
  - *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*
- *Example: Loop of length 16, 4 threads*

TID	0	1	2	3
no chunk	1-4	5-8	9-12	13-16
chunk = 2	1-2	3-4	5-6	7-8
	9-10	11-12	13-14	15-16

Ruud van der Pas (2005), "An introduction to OpenMP", IWOMP 2005, University of Oregon, Eugene, Oregon, USA

Compiler directives
for *construct*, schedule *clause*

- dynamic[, chunk]
  - *Fixed portions of work; size is controlled by the value of chunk.*
  - *When a thread finishes, it starts on the next portion of work.*
- guided[, chunk]
  - *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially.*
- runtime
  - *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE.*

Ruud van der Pas (2005), "An introduction to OpenMP", IWOMP 2005, University of Oregon, Eugene, Oregon, USA

Some experiment

Ruud van der Pas (2005), "An introduction to OpenMP", IWOMP 2005, University of Oregon, Eugene, Oregon, USA

*The **lastprivate** clause causes the variables listed being updated after the termination of the parallel region!*

```
13a.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  long i=7, sum=0;

  cout << "Before: i="<< i << endl;;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:sum) lastprivate(i)
    for(i=1;i<1000;i++) {
      sum += i;
    }
  }

  cout << "After: i=" << i << endl;
  return 0;
}
```

```
./13a.exe
Before: i=7
After: i=1000
```


*The **lastprivate** clause causes the variables listed being updated after the termination of the parallel region!*

```
13b.cpp
#include <iostream>
#include <omp.h>
using namespace std;

int main() {
  int i,j;
  #pragma omp parallel for schedule(static,2) collapse(2)
  for(i=0;i<5;i++) {
    for(j=0;j<3;j++) {
      int tid = omp_get_thread_num();
      cout << i << ", " << j << " - " << tid << endl;
    }
  }

  return 0;
}
```

```
0, 0 - 0
0, 1 - 0
0, 2 - 1
1, 0 - 1
1, 1 - 2
1, 2 - 2
2, 0 - 3
2, 1 - 3
2, 2 - 0
3, 0 - 0
3, 1 - 1
3, 2 - 1
4, 0 - 2
4, 1 - 2
4, 2 - 3
```

tid in this example is similar to rank in MPI!

Compiler directives
for *construct*, collapse *clause*

- If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space which is then divided according to the schedule clause.
- **The iteration count for each associated loop is computed before entry to the outermost loop**. If execution of any associated loop changes any of the values used to compute any of the iteration counts then the behavior is unspecified.
- Rules
  - Perfectly nested
  - Rectangular iteration space

```
for(int i=0;i<10;i++) {
    doSomething(i);
    for(int j=0;j<i;j++) {
        ….
    }
}
```

```
for(int i=0;i<10;i++) {
    for(int j=0;j<5;j++) {
        ….
    }
    i+=2;
}
```

•*The* **order** *clause causes the code region marked by "ordered" directive inside loops being ordered as if they were executed in sequential!*

•*Shortcut:*

#pragma omp parallel
#pragma omp for → ***#pragma omp parallel for***

```
14a.cpp
#include <iostream>

int main() {
  int i;
  #pragma omp parallel for schedule(static,4)
  for(i=0;i<100;i+=5) {
    std::cout << i << " ";
  }
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./14a.exe
0 5 10 15 80 85 90 95 40 45 50 55 20 25 30 35 60 65 70 75
```

```
OMP_NUM_THREADS=2 ./14a.exe
0 5 10 15 40 45 50 55 80 85 90 95 20 25 30 35 60 65 70 75
```

38

The **order** *clause causes the ordered region inside loops being executed in the order as if they were executed in sequential!*

```
15a.cpp
#include <iostream>

int main() {
  int i;
  #pragma omp parallel for schedule(static,4) ordered
  for(i=0;i<100;i+=5) {
    #pragma omp ordered
    std::cout << i << " ";
  }
  return 0;
}
```

```
OMP_NUM_THREADS=4 ./14a.exe
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

```
OMP_NUM_THREADS=2 ./14a.exe
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

39

Compiler directives
for *construct*, nowait *clause*

- At the end of the parallel for region, there is an implicit synchronization (barrier) point. **nowait** removes such barrier

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
                        (implied barrier)
```

Ruud van der Pas (2005), "An introduction to OpenMP", IWOMP 2005, University of Oregon, Eugene, Oregon, USA

40

Compiler directives
sections *construct*

#pragma omp sections *[clause [[,] clause] …] new-line*

{

*[***#pragma omp section** *newline]*

structured-block

*[***#pragma omp section** *newline]*

structured-block

…

}

Clauses

private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait

41

Each section defined is executed by exactly one thread!

```
16a.cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  #pragma omp parallel default(none) shared(cout)
  {
    int ID = omp_get_thread_num();
    cout << "ThreadID: " << ID << endl;
    #pragma omp sections
    {
      #pragma omp section
        for(int i=0;i<5;i++)
          cout << "(" << ID << " doing AI)" << endl;
      #pragma omp section
        for(int i=0;i<7;i++)
          cout << "(" << ID << " getting user inputs)" << endl;
      #pragma omp section
        for(int i=0;i<3;i++)
          cout << "(" << ID << " doing drawings)" << endl;
    }
  }
  return 0;
}
```

42

OMP_NUM_THREADS=6 ./16a.exe
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
ThreadID: 1
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
ThreadID: 5
ThreadID: 2
(2 doing drawings)
(2 doing drawings)
(2 doing drawings)
ThreadID: 3
ThreadID: 4

OMP_NUM_THREADS=2 ./16a.exe
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
(0 getting user inputs)
ThreadID: 1
(1 doing drawings)
(1 doing drawings)
(1 doing drawings)

OMP_NUM_THREADS=3 ./16a.exe
ThreadID: 0
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
(0 doing AI)
ThreadID: 2
(2 doing drawings)
(2 doing drawings)
(2 doing drawings)
ThreadID: 1
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)
(1 getting user inputs)

Compiler directives
single *construct*

#pragma omp single *[clause [[,] clause] …] new-line*
structured-block

Clauses
- *private(list)*
- *firstprivate(list)*
- *copyprivate(list)*
- *nowait*

- Exactly one unspecified thread executes the code in the single construct.
- There is an implicit barrier at the end of the single construct.
- Suitable for variable initialization, I/O, etc.

17a.cpp
```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  int i, n=10, sum=0;

  #pragma omp parallel default(none) \
       shared(cin, cout, n, sum) private(i)
  {
    int ID = omp_get_thread_num();
    #pragma omp single
    {
      cout << "(" << ID << ") Please enter n: ";
      cin >> n;
    }
    #pragma omp for reduction(+:sum)
    for(i=0;i<=n;i++) {
      sum += i;
    }
  }
  cout << "Sum: " << sum;
  return 0;
}
```


Summary

- Compiler Directives
  - Conditional compilation
  - Parallel construct
  - Work-sharing constructs
    - for, section, single
  - *Work-tasking*
  - *Synchronization*
- *Library Functions*
- *Environment Variables*