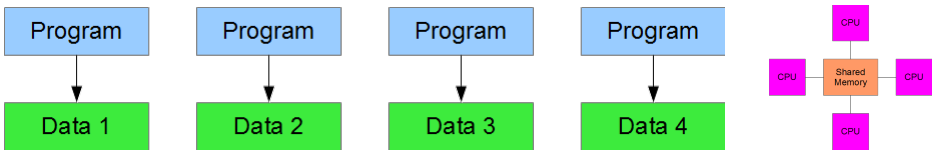
Part B - Languages

OpenMP - Fundamentals

Introduce directive based multi-threading
Show examples that implement the SPMD programming model

Introduction | Directives | Library | Environment | Examples | Exercises

OpenMP is an API specification for parallel programming solutions on shared memory architectures. OpenMP extensions to compilers implement an explicit SPMD programming model. That is, each thread executes the same set of instructions on different data. OpenMP stands for *Open Multi-Processing*.



Implementing the SPMD Programming Model on a Shared Memory Architecture

This chapter introduces OpenMP. The language extensions consist of compiler directives, library routines and environment variables. This chapter includes solutions to the problem of calculating the magnitude of a vector using different algorithms - two SPMD algorithms and one work-sharing algorithm.

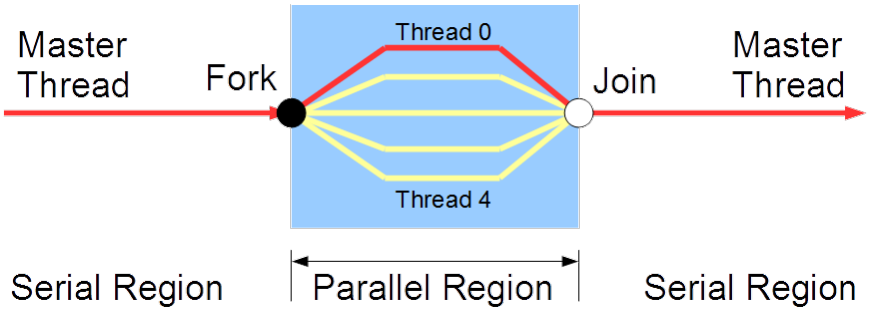
INTRODUCTION

The OpenMP standard was originally developed in 1994 (ANSI X3H5). An Architecture Review Board (ARB), which currently includes industry partners such as AMD, ARM, Cray, Fujtsu, HP, IBM, Intel, Micron, NEC, NVIDIA, Oracle Corporation, Red Hat, and Texas Instruments, has led its evolution. HPC computing centers are auxiliary members of the ARB. The first version of the OpenMP interface was published in 1997.

Michael Klemm, Matthijs van Waveren, and Jim Cownie have published a review of the first 20 years of evolution of OpenMP here

OpenMP

- is a compiler-directive-based extension to the core languages
- implements the fork-join parallel pattern: the master thread forks into a team of threads that execute in parallel and subsequently rejoin the master thread.



OpenMP Implements the Fork-Join Parallel Pattern

The OpenMP specification identifies three components (Source: Barney, B. (2014).):

1. compiler directives
2. runtime library routines

3. environment variables

There is a reference card for OpenMP C/C++ Syntax here.

COMPILER DIRECTIVES

OpenMP C/C++ compiler directives start with **`#pragma omp`**.

The directive follow the C/C++ standard conventions for compiler directives and end with a newline character. They serve as 'guidance' to the compiler.

We embed these directives in the source code. Each directive applies to the structured block that immediately follows the directive and consists of a construct and, optionally a set of clauses that qualify the construct:

```
#pragma omp construct [clause, ...] newline (\n)
structured block
```

The construct identifies the directive command and is mandatory. The square brackets identify the enclosed clauses as optional.

Constructs

An OpenMP parallel construct may be one of the following, amongst others:

- **`parallel`** - identifies a parallel region - a parallel region is a block of code that can be executed by multiple threads
- **`task`** - identifies a block of code as an explicit task to be executed by a single thread
- **`simd`** - identifies a loop that can be transformed into a SIMD loop
- Work Sharing Constructs
  - **`for`** - distributes iterations across the team of threads
  - **`sections`** - identifies a region that contains a set of structured blocks to be distributed among and executed by threads in a team
  - **`for simd`** - identifies a loop that can be transformed into a SIMD loop that will be executed in parallel by the threads in a team
  - **`single`** - identifies a structured block to be executed by a single thread in a team

  The **`for`** construct identifies a data parallelism solution. The **`sections`** construct identifies a task parallleism solution.
- Synchronization
  - **`master`** - identifies a region to be executed by the master thread only
  - **`critical`** - identifies a region to be executed by only one thread at a time
  - **`barrier`** - synchronizes all the threads in a team
  - **`taskwait`** - specifies a wait on the completion of the child tasks
  - **`taskgroup`** - specifies a wait on completion of child tasks of the current task, and waits for descendant tasks
  - **`atomic`** - identifies a memory location to be updated by one thread at a time
  - **`flush`** - establishes a consistent view of memory
  - **`ordered`** - executes in same order as on a serial processor
  - **`threadprivate`** - makes a global variable local and persistent to a thread through execution of multiple parallel regions

Shortcuts for combinations of these constructs include:

- **`parallel for`** - identifies a parallel region containing one or more associated loops and no other statements
- **`parallel sections`** - identifies a parallel region containing one sections construct and no other statements
- **`for simd`** - a loop that can be executed concurrently using simd instructions and can also be executed in parallel by the threads in the team
- **`parallel for simd`** - identifies a parallel region containing one for simd construct and no other statements

Clauses

The following clauses qualify a parallel command

- **`num_threads(integer)`** - sets the number of threads
- **`private(list)`** - lists the variables that are private to each thread
- **`shared(list)`** - lists the variables are shared amongst all threads in a team
- **`default(shared|none)`** - specifies the default scope for all variables in the lexical extent of the parallel region
- **`firstprivate(list)`** - lists the variables that are private to each thread but initialized to a serial value on entry
- **`lastprivate(list)`** - lists the variables the are private to each thread and copied to a serial value on exit
- **`collapse(integer)`** - identifies the number of loops to collapse into a single iteration
- **`linear(list:linear-step)`** - lists the variables that have a linear relationship to iteration space
- **`copyin(list)`** - assigns the same value to **`threadprivate`** variables for all threads in a team
- **`copyprivate(list)`** - broadcasts values acquired by a **`single`** thread to all instances of that private variable in all other threads
- **`reduction(operator: list)`** - performs the reduction specified by operator on the variables in the list
- **`if(boolean expression)`** - executes the command only if true
- **`ordered`** - iterations are to be executed in serial order
- **`nowait`** - threads do not synchronize at the end
- **`schedule(type [,chunk])`** - controls how iterations are distributed amongst the threads in a team

The **`schedule type`** may be one of:

- **`static`** - divide iterations into groups of size **`chunk`** and assign groups statically to the threads
- **`dynamic`** - divide iterations into groups of size **`chunk`** and assign groups dynamically to the threads as they become available
- **`guided`** - similar to dynamic, except block size decreases as more iterations are completed
- **`runtime`** - defer the type of scheduling until runtime
- **`auto`** - delegate the type of scheduling to the compiler or the runtime system

Which Clauses Qualify a Construct

The table below shows the clauses that each construct supports.

Clause	Construct									
	`parallel`	**`for`**	**`sections`**	**`single`**	**`simd`**	**`task`**	**`for simd`**	**`parallel`**		
								`for`	**`sections`**	**`for simd`**
`if`	x					x		x	x	x
`num_threads`	x							x	x	x
`private`	x	x	x	x	x	x	x	x	x	x
`shared`	x					x		x	x	x
`default`	x					x		x	x	x
`firstprivate`	x	x	x	x		x	x	x	x	x
`lastprivate`		x	x		x		x	x	x	x
`reduction`	x	x	x		x		x	x	x	x
`copyin`	x							x	x	x
`copyprivate`				x						
`collapse`		x			x		x	x		x
`linear`		x			x		x	x		x
`schedule`		x					x	x		x

ordered		x					x	x		x
nowait		x	x	x			x			

The following constructs do not accept clauses:

- master
- critical
- barrier
- taskwait
- taskgroup
- flush
- threadprivate

Example

In the following program each thread initializes its own private variable to the value of serial variable i declared outside the parallel region and displays the value of the private variable after the thread has incremented that value by 1:

```
// OpenMP - Parallel Construct
// omp_parallel.cpp

#include <iostream>
#include <omp.h>
                                                    i = 13
int main() {                                        i = 13
    int i = 12;                                     i = 13
                                                    i = 13
    #pragma omp parallel firstprivate(i)            i = 13
    {                                               i = 13
        std::cout << "\ni = " << ++i;               i = 13
    }                                               i = 13
    std::cout << "\ni = " << i << std::endl;        i = 12
}
```

Note that the value of i declared outside the parallel region is not modified.

RUNTIME LIBRARY ROUTINES

Runtime library routines affect and monitor threads, processors, and the parallel environment. These routines are external functions with "C" linkage.

These runtime library routines usually require the <omp.h> header file.

The runtime library routines include

- omp_set_num_threads(int) - sets number of threads in next parallel region
- omp_get_num_threads() - returns the number of threads from within the host team of threads
- omp_get_max_threads() - returns the maximum value from omp_get_num_threads()
- omp_get_thread_num() - returns the thread id within the team
- omp_get_thread_limit() - returns the maximum number of threads available to a program
- omp_get_num_procs() - returns the number of processors available to the program
- omp_in_parallel() - returns true if the number of nested parallel regions is greater than 0
- omp_set_dynamic(int) - controls dynamic adjustment of the number of threads available for execution of parallel regions
- omp_get_dynamic() - returns the state of dynamic thread adjustment
- omp_set_nested(int) - enables or disables nested parallelism

- `omp_get_nested()` - returns the state of nested parallelism
- `omp_set_schedule(omp_sched_t, int)` - sets loop scheduling policy if runtime scheduling is selected
- `omp_get_schedule(omp_sched_t*, int*)` - returns loop scheduling policy if runtime scheduling is selected
- `omp_set_max_active_levels(int)` - sets the maximum number of nested active parallel regions
- `omp_get_max_active_levels()` - returns the maximum number of nested active parallel regions
- `omp_get_level()` - returns the number of nested parallel regions (active or inactive) that enclose the call
- `omp_get_active_level()` - returns the number of nested active parallel regions that enclose the call
- `omp_in_final()` - returns true if in the final task region
- `omp_get_wtime()` - returns the wall clock time
- `omp_get_wtick()` - returns the double-precision floating-point value equal to the number of seconds between successive clock ticks

An active parallel region is one executed by a team consisting of more than one thread. An inactive parallel region is one that is executed by a team of only one thread.

Thread Number Example

```
// OpenMP - Runtime Routines
// omp_hi.cpp

#include <iostream>
#include <omp.h>

int main() {

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        std::cout << "Hi from thread "
         << tid << '\n';
    }
}
```

```
Hi from thread 0
Hi from thread 6
Hi from thread 2
Hi from thread 5
Hi from thread 7
Hi from thread 3
Hi from thread 1
Hi from thread 4
```

Output from different threads may intermingle across cascading points. Using `printf()` avoids any intermingling.

ENVIRONMENT VARIABLES

We can control the execution of an OpenMP program by prescribing background settings for the environment variables. The variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

These environment variables include

- `OMP_NUM_THREADS` - maximum number of threads to use
- `OMP_SCHEDULE` - how to schedule iterations on the processors
- `OMP_DYNAMIC` - controls dynamic adjustment of the number of threads available for execution
- `OMP_PROC_BIND` - whether to bind threads to processors
- `OMP_NESTED` - whether to allow nested parallelism
- `OMP_STACKSIZE` - default size of the stack for non-master threads
- `OMP_WAIT_POLICY` - desired behavior of waiting threads
- `OMP_MAX_ACTIVE_LEVELS` - maximum number of nested active parallel regions
- `OMP_THREAD_LIMIT` - number of threads to use for the whole program

Example

On Linux

```
sh/bash
export OMP_NUM_THREADS=8
```

```
csh/tcsh
setenv OMP_NUM_THREADS 8
```

At the Visual Studio command prompt:

```
set OMP_NUM_THREADS=8
```

EXAMPLES

Vector Magnitude

In this example, a vector is a set of values of identical type. Its magnitude or size is the square root of the sum of its elements squared.

Serial Version

The following serial program receives the size of a vector from the command line, initializes each element to a value of 1.0, squares each element, adds the squares together, and reports the square root of the sum. That is, the program calculates the magnitude of the vector:

```
// Vector Magnitude - Serial Version
// vec_mag_serial.cpp

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <omp.h>

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cerr <<
         "\n*** Incorrect no of arguments ***\n";
        std::cerr << argv[0] << " size\n";
        return 1;
    }
    int n = std::atoi(argv[1]);
    float* a = new float[n];
    for (int i = 0; i < n; i++)
        a[i] = 1.0f;
    float mag{ 0.f };

    double start = omp_get_wtime();

    for (int i = 0; i < n; i++)
        mag += a[i] * a[i];

    mag = std::sqrt(mag);
    std::cout << "|a| = " << mag <<
     "\n|a|*|a| = " << mag * mag <<
     "\nn = " << n << std::endl;
                                                    |a| = 3162.28
    double end = omp_get_wtime();                   |a|*|a| = 1e+07
    std::cout << "Single threaded." <<              n = 10000000
     "Time = " << end - start << "s." << std::endl; Single Threaded.
                                                    Time = 0.014298s.
    delete [] a;
}
```

SPMD Programming Model - Array Solution

The following parallel version executes the squaring and summing operations in parallel using the SPMD Programming Model, stores intermediate results for each thread in a separate array element and finally accumulates the squared element values to obtain the square of the vector's magnitude:

```
// Vector Magnitude - SPMD - Array Version
// vec_mag_SPMD.cpp

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <omp.h>

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cerr <<
         "\n*** Incorrect no of arguments ***\n";
        std::cerr << "Usage: " << argv[0] << " size\n";
        return 1;
    }
    int nthreads;
    int size = std::atoi(argv[1]);
    int mnt = omp_get_max_threads();
```

```
    float* a = new float[size];
    float* s = new float[mnt];
    for (int i = 0; i < size; i++)
        a[i] = 1.0f;

    double start = omp_get_wtime();

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nt  = omp_get_num_threads();
        if (tid == 0) nthreads = nt;
        s[tid] = 0.0f;
        for (int i = tid; i < size; i += nt)
            s[tid] += a[i] * a[i];
    }

    float mag = 0.0f;
    for (int i = 0; i < nthreads; i++)
        mag += s[i];
    mag = std::sqrt(mag);
    std::cout << "|a| = " << mag <<
     "\n|a|*|a| = " << mag * mag <<
     "\nn = " << size << std::endl;

    double end = omp_get_wtime();
    std::cout << mnt << " threads available\n" <<
     nthreads << " threads used.\nTime = " <<
     end - start << "s." << std::endl;

    delete [] a;
    delete [] s;
}
```

```
|a| = 3162.28
|a|*|a| = 1e+07
n = 10000000
8 threads available
8 threads used.
Time = 0.0114283s.
```

Before entering the parallel region this program determines the maximum number of threads available. Once in the parallel region, it determines the number of threads in the team (**nt**), which may differ from the maximum number of threads available. The parallel region partitions the vector into elements spaced **nt** apart. Each thread accumulates the squares of the elements in its partition. Outside the parallel region, the master thread accumulates the results for each partition.

SPMD Programming Model - Critical Version

The critical version avoids storing intermediate results for each thread in an array and summing the array elements outside the parallel region through use of the the **critical** construct, which ensures that only one thread updates **mag** at a time:

```
// Vector Magnitude - SPMD - Critical Version
// vec_mag_critical.cpp

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <omp.h>

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cerr <<
         "\n*** Incorrect no of arguments ***\n";
        std::cerr << "Usage: " << argv[0] << " size\n";
        return 1;
    }
    int nthreads;
    int size = std::atoi(argv[1]);
    int mnt = omp_get_max_threads();
```

```
    float* a = new float[size];
    for (int i = 0; i < size; i++)
        a[i] = 1.0f;
    float mag{0.0f};

    double start = omp_get_wtime();

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nt  = omp_get_num_threads();
        if (tid == 0) nthreads = nt;
        float sum = 0.0f;
        for (int i = tid; i < size; i += nt)
            sum += a[i] * a[i];
        #pragma omp critical
        mag += sum;
    }

    mag = std::sqrt(mag);
    std::cout << "|a| = " << mag <<
     "\n|a|*|a| = " << mag * mag <<
     "\nn = " << size << std::endl;

    double end = omp_get_wtime();
    std::cout << mnt << " threads available\n" <<
     nthreads << " threads used.\nTime = " <<
     end - start << "s." << std::endl;

    delete [] a;
}
```

```
|a| = 3162.28
|a|*|a| = 1e+07
n = 10000000
8 threads available
8 threads used.
Time = 0.0113094s.
```

In both SPMD examples, the program distributes the work equally across the active threads.

Work Sharing Version

The work-sharing version combines the **`parallel`** and **`for`** constructs with the **`reduction`** clause for a solution that is the most readable with closest-to-serial-syntax:

```
// Vector Magnitude - Work Sharing Version
// vec_mag_work_sharing.cpp

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <omp.h>

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cerr <<
         "\n*** Incorrect no of arguments ***\n";
        std::cerr << "Usage: " << argv[0] << " size\n";
        return 1;
    }
    int nthreads;
    int size = std::atoi(argv[1]);
    int mnt = omp_get_max_threads();
    float* a = new float[size];
    for (int i = 0; i < size; i++)
        a[i] = 1.0f;
    float mag{ 0.f };

    double start = omp_get_wtime();
```

```
    #pragma omp parallel
    {
        #pragma omp for reduction(+:mag)
        for (int i = 0; i < size; i++)
            mag += a[i] * a[i];
        #pragma omp single
        nthreads = omp_get_num_threads();
    }

    mag = std::sqrt(mag);
    std::cout << "|a| = " << mag <<
     "\n|a|*|a| = " << mag * mag <<
     "\nn = " << size << std::endl;

    double end = omp_get_wtime();
    std::cout << mnt << " threads available\n" <<
     nthreads << " threads used.\nTime = " <<
     end - start << "s." << std::endl;

    delete [] a;
}
```

```
|a| = 3162.28
|a|*|a| = 1e+07
n = 10000000
8 threads available
8 threads used.
Time = 0.0092028s.
```

Timing Statistics

The execution times for the different solutions using various compilers are listed in the table below

Compiler	Serial	SPMD Array	SPMD Critical	Work Sharing
Visual Studio 2019 (8 threads)	0.0142980	0.0114283	0.0113090	0.0092028
Intel C++ 19.1 (8 threads)	0.0146281	0.0087898	0.0092941	0.0070684
GCC 9.1 (4 threads)	0.0132333	0.0057774	0.0056613	0.0036689

LLNL Examples

Examples from Barney, B. (2018). modified to C++ versions:

- omp_hello.cpp - uses `parallel private omp_get_thread_num(), omp_get_num_threads()`
- omp_getEnvInfo.cpp - uses `parallel private omp_get_...`
- omp_workshare1.cpp - uses `parallel shared private for schedule`
- omp_workshare2.cpp - uses `parallel shared private sections nowait section`
- omp_reduction.cpp - uses `parallel for reduction`
- omp_orphan.cpp - uses `parallel for reduction omp_get_thread_num()`
- omp_mm.cpp - uses `parallel shared private for schedule`

EXERCISES

- Barney, B. (2018). "OpenMP". Lawrence Livermore National Laboratory. Retrieved June 28 2018. Last Modified June 28 2018.
- Mattson, T. (2013). Tutorial Overview | Introduction to OpenMP
- Microsoft MSDN (2020).OpenMP in Visual C++