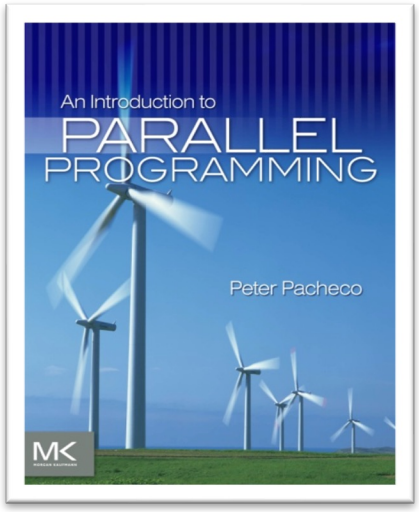
Introduction to Parallel Programming II

Center for Institutional Research Computing



Slides for the book "An introduction to Parallel Programming", by Peter Pacheco (available from the publisher website): http://booksite.elsevier.com/9780123742605/

OpenMP for shared memory multithreaded programming

- OpenMP is an implementation of multithreading

OpenMP for shared memory multithreaded programming

- OpenMP is an implementation of multithreading
- A primary thread (a series of instructions executed consecutively)

OpenMP for shared memory multithreaded programming

- OpenMP is an implementation of multithreading
- A primary thread (a series of instructions executed consecutively)
- A number of sub-threads

OpenMP for shared memory multithreaded programming

- OpenMP is an implementation of multithreading
- A primary thread (a series of instructions executed consecutively)
- A number of sub-threads
  - Forked from the primary thread
  - The system divides a task among them

OpenMP for shared memory multithreaded programming

- OpenMP is an implementation of multithreading
- A primary thread (a series of instructions executed consecutively)
- A number of sub-threads
  - Forked from the primary thread
  - The system divides a task among them
- The threads then run concurrently

Pragmas

- Special preprocessor instructions.

#pragma

Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C/C++ specification.

#pragma

Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C/C++ specification.
- Compilers that don't support the pragmas ignore them.

#pragma

OpenMp pragmas

- # pragma omp parallel

- # include omp.h

OpenMp pragmas

- # pragma omp parallel

- # include omp.h

  – Most basic parallel directive.

OpenMp pragmas

- # pragma omp parallel

- # include omp.h

  – Most basic parallel directive.
  – The number of threads that run the following structured block of code is determined by the run-time system.

clause

- Text that modifies a directive.

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

clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.

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

clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

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

Some terminology

- In OpenMP, the collection of threads executing the parallel block — the original thread and the new threads — is called a team, the original thread is called the master, and the additional threads are called worker.

Parallel Code Template (OpenMP)

```
#include <omp.h>

main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel
{
…. // openmp parallel region where p threads are active and running concurrently
}
```

Parallel Code Template (OpenMP)

```
#include <omp.h>

main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel
{
…. // openmp parallel region where p threads are active and running concurrently
}
```

Parallel Code Template (OpenMP)

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

```
main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel
{
…. // openmp parallel region where p threads are active and running concurrently
}
```

Parallel Code Template (OpenMP)

```
#include <omp.h>

main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel
{
…. // openmp parallel region where p threads are active and running concurrently
}
```

Parallel Code Template (OpenMP)

```
#include <omp.h>

main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel
{
…. // openmp parallel region where p threads are active and running concurrently
}
```

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

Serial version of “hello world”

```
#include <stdio.h>

int main()
{
   printf("Hello world\n");
   return 0;
}
```

Compile it: g++ hello.cpp

How do we invoke omp in this case?

Serial version of “hello world”

```
#include <stdio.h>

int main()
{
   printf("Hello world\n");
   return 0;
}
```

Compile it: g++ hello.cpp

How do we invoke omp in this case?

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

This is the parallel region

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

This is the parallel region

Compile it: gcc –fopenmp hello.c

After invoking omp

```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <thread>
#include <iostream>

void hello_world() {
    // Get the total number of cores in the system
    const auto processor_count = std::thread::hardware_concurrency();
    // Set the number of threads used in OpenMP
    omp_set_num_threads(processor_count);
    // Begin of parallel region
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        printf( format: "Welcome from thread = %d\n",
               tid);

        if (tid == 0) {

            // Only master thread does this
            int nthreads = omp_get_num_threads();
            printf( format: "Number of threads = %d\n",
                   nthreads);
        }
    }
}
```

This is the parallel region

Compile it: gcc –fopenmp hello.c
CMakeLists.txt: add Flag '-fopenmp'

Add openmp flag in cmake

- Add openmp flag in your CMakeLists.txt
- set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp")

Sum using OpenMP

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

Partition the data

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Set num of threads

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Set num of threads

Use local sum instead of sum to avoid resource competition on the sum variable

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Set num of threads

Use local sum instead of sum to avoid resource competition on the sum variable

Parallelize the task

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Set num of threads

Use local sum instead of sum to avoid resource competition on the sum variable

Parallelize the task

Aggregate the local result to the global sum

Sum using OpenMP

sum is in the shared scope. All threads can access it.
Similarly, the data[] is also shared.

```
int cal_sum(){
    int sum = 0;
    int data_size = 100000000;
    int num_threads = 2;
    int workload = data_size/num_threads;
    int* data = new int[data_size];
    for (int i = 0; i < data_size; ++i) {
        data[i] = 1;
    }
    omp_set_num_threads(num_threads);
    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();
#pragma omp parallel
    {
        // Getting thread number
        int tid = omp_get_thread_num();
        int local_sum = 0;
        for (int i = tid*workload; i < (tid + 1)*workload; ++i) {
            local_sum += data[i];
        }
        sum += local_sum;
    }
    std::cout<< sum << std::endl;
    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
    std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count()
    << "[microseconds]" << std::endl;
}
```

Partition the data

Initialize the data

Set num of threads

Use local sum instead of sum to avoid resource competition on the sum variable

parallel region

Parallelize the task

Aggregate the local result to the global sum