Unit 19

OpenMP Library for Parallelism
Overview of OpenMP

• A library or API (Application Programming Interface) for parallelism

• Requires compiler support (make sure the compiler you use supports OpenMP)
  – g++, clang++, MSVC all support OpenMP
  – Enable support in gcc/g++/clang++ using: -fopenmp
    • g++ -fopenmp test_omp.cpp -o test_omp

• In your code:
  – Functions (prototyped in #include<omp.h> )
  – Compiler Directives (#pragma ...)


Using OpenMP

**Base, Sequential Program**

```c
const int MAX = 100000;
int data[MAX];

void init()
{
    for(int i=0; i < MAX; i++)
    {
        data[i] = 0;
    }
}

int main()
{
    init();
    /* Use initialized array */
    return 0;
}
```

**Fictitious "Manual" Parallelization**

```c
const int MAX = 100000;
int data[MAX];

void init(int s, int n)
{
    for(int i=s; i < s+n; i++)
    {
        data[i] = 0;
    }
}

int main()
{
    const int T = /* # of threads */;
    int n = MAX/T;
    for(int i=0; i < T; i++)
    {
        create_thread(init(i*n, n));
    }
    /* Let parallel work happen */
    for(int i=0; i < T; i++)
    {
        // wait for threads to finish
        wait_for_thread(i);
    }
    /* Use initialized array */
    return 0;
}
```

**Parallel Program Using OpenMP**

```c
const int MAX = 100000;
int data[MAX];

void init()
{
    #pragma omp parallel for
    for(int i=0; i < MAX; i++)
    {
        data[i] = 0;
    }
}

int main()
{
    init();
    /* Use initialized array */
    return 0;
}
```
Parallel for

- Place `#pragma omp parallel for` before the actual 'for' loop
- OpenMP "automatically"...
  - Adds code to determine how many threads to create
  - Creates ("forks") the threads
  - Determines which iterations of the for loop will be handled by each thread
  - Waits for ("joins") each thread to finish

```c
const int MAX = 100000;
int data[MAX];

void init()
{
    #pragma omp parallel for
    for(int i=0; i < MAX; i++)
    {
        data[i] = 0;
    }
}

int main()
{
    init();
    /* Use initialized array */
    return 0;
}
```

Thread Creation/Fork...

Split iterations of the for loop amongst the various threads

Thread Wait/Join...

Before 'for'

T1  T2  TN

After 'for'
Rules for Using Parallel for

- `#pragma omp parallel for`

**Rules:**
- Must have an ‘int’ as loop counter
- Termination condition must not be changed inside the loop
- No break statements allowed in the loop

**Why?** So that the compiler can figure out how many iterations each thread should execute at the start of the loop

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

int a[1000000];

int main()
{
  int i, x = 1000000;
  // Good!
  #pragma omp parallel for
  for(i=0; i < x; i++){
    a[i] = 0;
  }

  #pragma omp parallel for
  for(i=0; i < x; i++){
    if(a[i] < 0)
      x--; // Changes # of iters.
  }

  // Bad!
  #pragma omp parallel for
  for(i=0; i<1000000; i++){
    if(a[i] < 0)
      break; // No break statements
  }

  return 0;
}
The Overhead of Threads

• There is a fairly significant overhead of creating and waiting for threads so to make it worthwhile there must be either:
  – A LOT of iterations
  – Or significant amount of work per iteration

• Demo:
  – init and init-par with 1, 2, 4 threads using array sizes of
    • 1 million
    • 10 million
    • 100 million

```c
// Is this enough iterations? const int MAX1 = 10000;
// Is this enough iterations? const int MAX2 = 1000000000;
int data[MAX1];

void init()
{
    #pragma omp parallel for
    for(int i=0; i < MAX1; i++){
        data[i] = 0;
    }
}

int main()
{
    init();
    /* Use initialized array */
    return 0;
}
```
Parallel Summation

```
const int MAX = 1000000;
int data[MAX];

int do_sum(int s, int n)
{
    int lsum = 0;  // private=per thread
    for(int i=s; i < s+n; i++){
        lsum += data[i];  // does this need
                           // to be atomic? No!
    }
    return lsum;
}

int main()
{
    const int T = /* # of threads */;
    int n = MAX/T;  /* items per thread */
    int sum = 0;
    for(int i=0; i < T; i++){
        create_thread(do_sum(i*n, n));
    }

    /* Let parallel work happen */
    for(int i=0; i < T; i++){
        // get returned lsums from threads
        sum += wait_for_thread(i);
    }
    cout << sum << endl;
    return 0;
}
```

Fictitious "Manual" Parallelization

```
const int MAX = 1000000;
int data[MAX];

int sum()
{
    int total = 0;
    #pragma omp parallel for reduction(+:total)
    for(int i=0; i < MAX; i++){
        total += data[i];
    }
    return total;
}

int main()
{
    /* Init array somehow */
    int mysum = sum();
    cout << mysum << endl;
    return 0;
}
```

Parallel Program Using OpenMP
Reductions

• OpenMP automatically handles reductions by simply indicating which variable should be combined and by which operator.

• OpenMP will make private versions of the variable for each thread and then combine the results into the reduction variable as the threads finish.

• Rules:
  − Variable must exist before 'for' loop starts and be updated inside.
  − Operators: +,-,*,&,|,&&,||

```c
const int MAX = 1000000;
int data[MAX];

int sum()
{
    int total = 0;
    #pragma omp parallel for reduction(+:total)
    for(int i=0; i < MAX; i++){
        total += data[i];
    }
    return total;
}

bool allOdd()
{
    bool odd = true;
    #pragma omp parallel for reduction(&&:odd)
    for(int i=0; i < MAX; i++){
        odd = odd && (data[i] % 2 == 1);
    }
}

bool search(int target)
{
    bool exists = false;
    #pragma omp parallel for reduction(||:exists)
    for(int i=0; i < MAX; i++){
        exists = exists || (data[i] == target);
    }
}
```
Sum Demo

- sum-par with 1, 2, and 4 threads for array sizes
  - 1 million
  - 10 million
  - 100 million
- Remove the 'reduction' clause
  - Different results compared to sequential version
  - Takes a lot longer (additional memory traffic due to single shared variable)

```c
const int MAX = 1000000;
int data[MAX];

int sum()
{
    int total = 0;
    #pragma omp parallel for reduction(+:total)
    for(int i=0; i < MAX; i++)
    {
        total += data[i];
    }
    return total;
}

int main()
{
    /* Init array somehow */
    int mysum = sum();
    cout << mysum << endl;
    return 0;
}
```
Prime Number Check Demos

- Problem: Determine how many numbers in the range \([m \text{ to } n]\) are prime
- primes-par with 1, 2, and 4 threads for ranges:
  - 1 to 10,000
  - 1 to 100,000
  - 1 to 500,000
- Remove the 'private' clause
  - Different results due to shared access
  - Takes a lot longer (additional memory traffic to single shared variable)
  - Could add '#pragma omp critical' around entire shared usage to enforce "one-at-a-time"

```cpp
int primes(int m, int n)
{
    int total = 0;
    bool isprime = true;
    #pragma omp parallel for private(isprime) reduction(+:total)
    for(int i=m; i <= n; i++){
        isprime = true;
        for(int j = 2; j < sqrt(i)+1; j++){
            if( i % j == 0){
                isprime = false;
            }
        }
        if(isprime) total++;
    }
    return total;
}
```

<table>
<thead>
<tr>
<th></th>
<th>isprime:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td>T2</td>
</tr>
<tr>
<td>i=5, j=2</td>
<td></td>
<td>i=1005 iter.</td>
</tr>
<tr>
<td>if(i%j == 0)</td>
<td>F</td>
<td>i++ (i=1006)</td>
</tr>
<tr>
<td>Isprime = false</td>
<td></td>
<td>isprime=true</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>if(isprime)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRONG!</td>
</tr>
</tbody>
</table>
Shared vs. Private Variables

- **Shared Variables**
  - Single copy shared between threads
  - Can suffer from synchronization (atomic access) issues
  - Variables declared before the parallel section begins are shared by default

- **Private Variables**
  - Separate copies made for each thread
  - Loop counter is private by default
  - Variables declared inside parallel loop are automatically private

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

int a[100][100], b[100][100];
int c[100][100];

int main(int argc, char *argv[])
{
    int i,j,k,temp,x;
    // i is private
    // j,k,temp, a[],b[],c[] are shared
    #pragma omp parallel for
    for(i=0; i < 100; i++){
        for(j=0; j < 100; j++){
            temp = 0;
            for(k=0; k < 100; k++){
                temp += a[i][k]*b[k][j];
            }
            C[i][j] = temp;
        }
    }
    return 0;
}
```

This code will produce errors due to the fact that the threads will stomp on each others’ values of j,k,temp.

![Parallel Matrix Multiply](image)
Shared vs. Private Variables

- **Private Variables**
  - Separate copies made for each thread
  - Loop counter is private by default
  - Variables declared inside parallel loop are automatically private

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

int a[100][100], b[100][100];
int c[100][100];

int main(int argc, char *argv[])
{
    int i;
    // i,j,k,temp are private
    // a[], b[], c[] are shared
    #pragma omp parallel for
    for(i=0; i < 100; i++)
    {
        for(int j=0; j < 100; j++)
        {
            int temp = 0;
            for(int k=0; k < 100; k++)
            {
                temp += a[i][k]*b[k][j];
            }
            C[i][j] = temp;
        }
    }
    return 0;
}
```

Variables declared inside the parallel for will be treated as private by default. This code will now work correctly.

---

**Parallel Matrix Multiply**
Specifying Shared or Private

• After ‘#pragma omp parallel for’ you can explicitly indicate shared vs. private variables via a ‘shared’ or ‘private’ clause
  – shared(var1, var2, ..., varN)
  – private(var1, var2, ..., varM)
• Used shared variables if they are read-only or write-only (i.e. a & b & c matrix)
• Make internal loop counters, temporary, and other independent variables private

We can explicitly define private variables through the private clause

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

int a[100][100], b[100][100];
int c[100][100];

int main(int argc, char *argv[])
{
    int i, j, k, temp;
    // i, j, k, temp are private
    // a[], b[], c[] are shared
    #pragma omp parallel for private(j,k,temp)
    for(i=0; i < 100; i++){
        for(j=0; j < 100; j++){
            temp = 0;
            for(k=0; k < 100; k++){
                temp += a[i][k]*b[k][j];
            }
            c[i][j] = temp;
        }
    }
    return 0;
}
```
Tips

• Declare any variables that **DO NOT** need to be shared **INSIDE** the parallel for loop to make it private by default

• Anything that is **SHARED** must be **read-only** (or write-only)

  – If you are **both reading and updating** a shared variable inside your parallel loop (other than the reduction variable) you will get **INCORRECT** results
Other Clauses

- Other clauses (like shared, private, reduction)
  - `firstprivate(vars)`: private variable for each thread but all copies will be initialized with the value of the variable before the threads were created
  - `if(expr)`: parallelize the following section only if expr evaluates to true
    - For example, we could check whether the size of an array is large enough to make parallelization useful
  - `schedule()`: how to distribute work to the threads
    - Statically = each thread gets equal portion of loop iterations/work
    - Dynamically = give each thread more work once it finishes current chunk of work
  - More...
Parallel Tasks

- Used for sections of different code that can be executed concurrently
- Use `#pragma omp parallel sections`
  - Code can be broken into sections which can be executed in parallel
  - `#pragma omp section { ... }` defines a parallel section of code
- Implicitly waits until all sections have finished executing before continuing past the parallel sections

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

int a[1000000], b[1000000];

int main(int argc, char *argv[])
{
    int i;
    #pragma omp sections private(i)
    {
        #pragma omp section
        {
            for(i=0; i<1000000; i++){
                a[i] = 0;
            }
        }
        #pragma omp section
        {
            for(i=0; i<1000000; i++){
                b[i] = 0;
            }
        }
        // won't execute here until both sections are finished
        return 0;
    }
}
```

Both loops will be performed at the same time
Other Functions

• Can call these functions in your code to control parallel operation
  – `int omp_get_num_threads()`
    • get the number of threads used in a parallel region
  – `int omp_get_thread_num()`:
    • get the thread ID/rank in a parallel region
    • Returns a unique value between 0 and T-1 to each thread
      (where T is the number of threads created by the library in the parallel region)
  – `void omp_set_num_threads(int nthreads)`
    • set the number of threads used in a parallel region
Other Features

• OpenMP includes support/functions for common parallel programming concepts
  – Barrier – Check-in point for all threads (no thread can continue until all others reach the barrier)
  – Critical Section (only 1 thread can be executing code in the critical section at a time)
  – Atomic (read-update-write operation is performed all at once...i.e. atomically)
  – Locks, Scheduling, and other features...
References

Reference Cards

Tutorials
• https://computing.llnl.gov/tutorials/openMP/
• http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
• http://bisqwit.iki.fi/story/howto/openmp/
OTHER EXAMPLES OF PARALLELISM
Graphics Representation

- A graphic is just a 2-D array of pixel values
- Pixel color represented as a numbers (i.e. 0 = black, 255 = white)

Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston
Graphics Operations

• **Brightness**
  - Each pixel value is increased/decreased by a constant amount
  - \( P_{\text{new}} = P_{\text{old}} + B \)
    - \( B > 0 \) = brighter
    - \( B < 0 \) = less bright

• **Contrast**
  - Each pixel value is multiplied by a constant amount
  - \( P_{\text{new}} = C \times P_{\text{old}} \)
    - \( C > 1 \) = more contrast
    - \( 0 < C < 1 \) = less contrast
More Graphics Operations

- Filter Effects like smoothing, blurring, etc. are produced by allowing a pixel’s value to be influenced by surrounding pixels (i.e. 2D-Average)
- Example: \( P_{\text{new}} = \frac{1}{9} \times [P + \Sigma(\text{neighbor pixels})] \)

We call this 2D array of weights the kernel
# More Graphics Operations

<table>
<thead>
<tr>
<th>P_{NW}</th>
<th>P_N</th>
<th>P_{NE}</th>
<th>[\frac{1}{9}]</th>
<th>[\frac{1}{9}]</th>
<th>[\frac{1}{9}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{W}</td>
<td>P</td>
<td>P_{E}</td>
<td>[\frac{1}{9}]</td>
<td>[\frac{1}{9}]</td>
<td>[\frac{1}{9}]</td>
</tr>
<tr>
<td>P_{SW}</td>
<td>P_S</td>
<td>P_{SE}</td>
<td>[\frac{1}{9}]</td>
<td>[\frac{1}{9}]</td>
<td>[\frac{1}{9}]</td>
</tr>
</tbody>
</table>

3x3 Blurring kernel

<table>
<thead>
<tr>
<th>1/25</th>
<th>1/25</th>
<th>1/25</th>
<th>1/25</th>
<th>1/25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
</tr>
<tr>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
</tr>
<tr>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
</tr>
<tr>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
<td>1/25</td>
</tr>
</tbody>
</table>

5x5 Blurring kernel

**Images:**

- **Original**
- **Smooth**
- **Smoother**
Parallel Tasks

• We need to apply the kernel to ALL pixels in the image
  – These require massive for loops

```
#include<iostream>

unsigned char iimage[SIZE][SIZE][RGB];
unsigned char oimage[SIZE][SIZE][RGB];
double kernel[N][N]

int main(int argc, char *argv[]) {
  ...
  // Apply the NxN kernel to the SIZE x SIZE image
  for(int y=N/2;y<SIZE+N/2;y++) // row of image
    for(int x=N/2;x<SIZE+N/2;x++) // column of image
      for(int k=0;k<RGB;k++)      // Red/Green/Blue plane
        for(int i=0; i<N; i++)    // row of kernel
          for(int j=0; j<N; j++)   // col of kernel
            oimage[y-(N/2)][x-(N/2)][k] += iimage[y+i-N/2][x+j-N/2][k]*kernel[i][j];
  return 0;
}
```

Many image and video effects are performed by applying various weights of an NxN kernel