

Lecture 7

OpenMP: pitfalls

Race condition

Data Dependence

Deadlock

SMP Programming errors

- Shared memory parallel programming is a mixed bag:
 - ◆ It saves the programmer from having to map data onto multiple processors. In this sense, its much easier.
 - ◆ It opens up a range of new errors coming from unanticipated shared resource conflicts.

2 major SMP errors

- Race Conditions

- The outcome of a program depends on the detailed timing of the threads in the team.

- Deadlock

- Threads lock up waiting on a locked resource that will never become free.

Race Conditions

```
#pragma omp parallel sections  
{  
  A = B + C  
  #pragma omp section  
  B = A + C  
  #pragma omp section  
  C = B + A  
}
```

- The result varies unpredictably based on detailed order of execution for each section.
- Wrong answers produced without warning!

Race Conditions:

A complicated solution

```
int ICOUNT = 0;
#pragma omp parallel sections
{
    A = B + C;
    ICOUNT = 1;
    #pragma omp flush(ICOUNT)
    #pragma omp section
    while (ICOUNT < 1)
        #pragma omp flush(ICOUNT)
    B = A + C;
    ICOUNT = 2;
    #pragma omp flush(ICOUNT)
    #pragma omp section
    while (ICOUNT < 2)
        #pragma omp flush(ICOUNT)
    B = A + C;
}
```

- In this example, we choose the assignments to occur in the order A, B, C.
 - ◆ ICOUNT forces this order.
 - ◆ FLUSH so each thread sees updates to ICOUNT - NOTE: you need the flush on each read and each write.

Examples: Race Conditions

```
#pragma omp parallel shared(x) private(tmp)
{
    int id = omp_get_thread_num();
    #pragma omp for reduction(+:x) nowait
        for (int j=0; j<100; j++) {
            tmp = work(j);
            x = x + tmp;}
    y[id] = more_work(x,id);
}
```

- The result varies unpredictably because the value of x isn't correct until the barrier at the end of the for loop is reached.
- Wrong answers are produced without warning!
- Be careful when using `nowait`!

Examples: Race Conditions (cont.)

```
float tmp;  
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    #pragma omp for reduction(+:x)  
    for (int j=0; j<100; j++) {  
        tmp = work(j);  
        x = x + tmp;}  
    y[id] = more_work(x,id);  
}
```

- The result varies unpredictably because access to the shared variable `tmp` is not protected.
- Wrong answers are produced without warning!
- Probably want to make `tmp` private.

OpenMP PI Program:

private clause and a critical section

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 16

int main() {
    double x, pi=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x)
    {
        int id = omp_get_thread_num();
        double sum = 0.0;
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x); }
        #pragma omp critical
            pi += sum;
    }
    pi=pi*step;
    printf("PI = %16.14f\n",pi);
}
```

Race conditions and the “pi” program

- Return to your “pi” program and this time, drop the private clause on x. In other words, let all threads use the same global variable for x.
 - ◆ Does your program still work?
 - ◆ Run it many times and see what happens to the answer.
 - ◆ Change the number of threads. Does the answer change?

</home/syam/ces745/openmp/C/pi/DEMO/pi-critical2.c>

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.139293
```

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.139667
```

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.139493
```

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.140276
```

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.137184
```

```
[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
```

```
PI = 3.140468
```



Data Dependences

Removal

A loop containing multiple data dependencies

```
    for (i=2; i<n-1; i++) {  
10    x = d[i] + i;  
20    a[i] = a[i + 1] + x;  
30    b[i] = b[i] + b[i - 1] + d[i - 1];  
40    c[2] = 2 * i;  
    }
```

Memory location	Line	Iteration earlier	Access	Line	Iteration later	Access	Loop carried	Kind of dependence
x	10	i	w	20	i	r	n	flow
x	10	i	w	10	i+1	w	y	output
x	20	i	r	10	i+1	w	y	anti
a[i+1]	20	i	r	20	i+1	w	y	anti
b[i]	30	i	w	30	i+1	r	y	flow
c[2]	40	i	w	40	i+1	w	y	output

Remove dependencies

- removal of anti dependencies

Serial version containing anti dependencies

// Array a is assigned before start of loop

```
for (i=0; i<n-1; i++) {  
    x = (b[i] + c[i])/2;  
    a[i] = a[i+1] +x;  
}
```

Parallel version with dependencies removed

```
#pragma omp parallel for shared(a, a2)
```

```
for (i=0; i<n-1; i++) {
```

```
    a2[i] = a[i+1];
```

- make a copy of the array

```
}
```

```
#pragma omp parallel for shared(a, a2) private(x)
```

```
for (i=0; i<n-1; i++) {
```

```
    x = (b[i] + c[i])/2;
```

```
    a[i] = a2[i] +x;
```

```
}
```

Remove dependencies

- **removal of output dependencies**

Serial version containing output dependencies

```
for (i=0; i<n; i++) {  
    x = (b[i] + c[i])/2;  
    a[i] = a[i] + x;  
    d[1] = 2 * x;  
}  
y = x + d[1] + d[2];
```

Parallel version with dependencies removed

```
#pragma omp parallel for shared(a) lastprivate(x, d1)  
for (i=0; i<n; i++) {  
    x = (b[i] + c[i])/2;  
    a[i] = a[i] + x;  
    d1 = 2 * x;  
}  
d[1] = d1;  
y = x + d[1] + d[2];
```

Remove dependencies

- **removal of flow dependencies caused by a reduction**

Serial version containing a flow dependence

```
x = 0;
for (i=0; i<n; i++) {
    x = x + a[i];
}
```

Parallel version with dependencies removed by reduction clause

```
x = 0;
#pragma omp parallel for reduction(+: x)
for (i=0; i<n; i++) {
    x = x + a[i];
}
```


Remove dependencies

- removal of flow dependencies using loop skewing

Serial version containing a loop-carried flow dependence from assignment to $a[i]$ at line 20 in iteration i to the read of $a[i-1]$ at line 10 in iteration $i+1$

```
for (i=1; i<n; i++) {  
10    b[i] = b[i] + a[i-1];  
20    a[i] = a[i] + c[i];  
}
```

Convert a loop-carried flow dependency into a non-loop-carried one

Line 20 can be computed in parallel because its value does not depend on other elements of a . We can shift, or ‘skew’, the subsequent read of $a[i]$ from iteration $i+1$ to iteration i , so that the dependency becomes non-loop-carried.

By adjusting subscripts and loop bounds appropriately, we have

```
b[1] = b[1] + a[0];  
#pragma omp parallel for shared(a, b, c)  
for (i=1; i<n; i++) {  
20    a[i] = a[i] + c[i];  
10    b[i+1] = b[i+1] + a[i];  
}  
a[n-1] = a[n-1] + c[n-1];
```

Dealing with non-removable dependencies

- **parallelization of a loop nest containing a recurrence**

Serial version containing a recurrence

```
for (j=1; j<n; j++) {  
    for (i=0; i<n; i++) {  
        a[j][i] = a[j][i] + a[j-1][i];  
    }  
}
```

Parallel version to the loop in the nest

```
for (j=1; j<n; j++) {  
#pragma omp parallel for shared (a)  
    for (i=0; i<n; i++) {  
        a[j][i] = a[j][i] + a[j-1][i];  
    }  
}
```

Dealing with non-removable dependencies

- **parallelization of part of a loop using fissioning**

Serial version containing a recurrence

```
    for (j=1; j<n; j++) {  
10      a[j] = a[j] + a[j-1];  
20      y = y + c[j];  
    }
```

Parallel version

```
    for (j=1; j<n; j++) {  
10      a[j] = a[j] + a[j-1];  
    }
```

```
#pragma omp parallel for reduction(+: y)  
    for (j=1; j<n; j++) {  
20      y = y + c[j];  
    }
```

Parallelizing an inner loop with Backward dependency

```
for (iter=0; iter<numiter; iter++) {  
    for (i=0; i<size-1; i++) {  
        V[i] = f( V[i], V[i+1] );  
    }  
}
```

```
/* 3. ITERATIONS LOOP */  
for (iter=0; iter<numiter; iter++) {  
    /* 3.1. DUPLICATE THE FULL ARRAY IN PARALLEL */  
    #pragma omp parallel for default(none) shared(V,oldV,totalSize) private(i) schedule(static)  
    for (i=0; i<totalSize; i++) {  
        oldV[i] = V[i];  
    }  
    /* 3.2. INNER LOOP: PROCESS ELEMENTS IN PARALLEL */  
    #pragma omp parallel for default(none) shared(V,oldV,totalSize) private(i) schedule(static)  
    for (i=0; i<totalSize-1; i++) {  
        V[i] = f(V[i],oldV[i+1]);  
    }  
}
```

Examples: Deadlock

```
omp_init_lock(&lcka);  
omp_init_lock(&lckb);  
#pragma omp parallel sections  
{  
    omp_set_lock(lcka);  
    omp_set_lock(lckb);  
    useAandB();  
    omp_unset_lock(lcka);  
    omp_unset_lock(lckb);  
    #pragma omp section  
    omp_set_lock(lckb);  
    omp_set_lock(lcka);  
    useBandA();  
    omp_unset_lock(lckb);  
    omp_unset_lock(lcka);  
}
```

- If A is locked by one thread and B by another, you have deadlock.
- Avoid nesting different locks.

Examples: Deadlock (cont.)

```
    omp_init_lock(&lcka);  
#pragma omp parallel sections  
{  
    omp_set_lock(lcka);  
    ival = work();  
    if (ival == tol)  
        omp_unset_lock(lcka);  
    else  
        error(ival);  
#pragma omp section  
    omp_set_lock(lcka);  
    useBandA();  
    omp_unset_lock(lcka);  
}
```

- If A is locked in the first section and the if statement branches around the unset lock, then threads in the other section will deadlock waiting for the lock to be released.
- Make sure you release your locks!

Conclusion

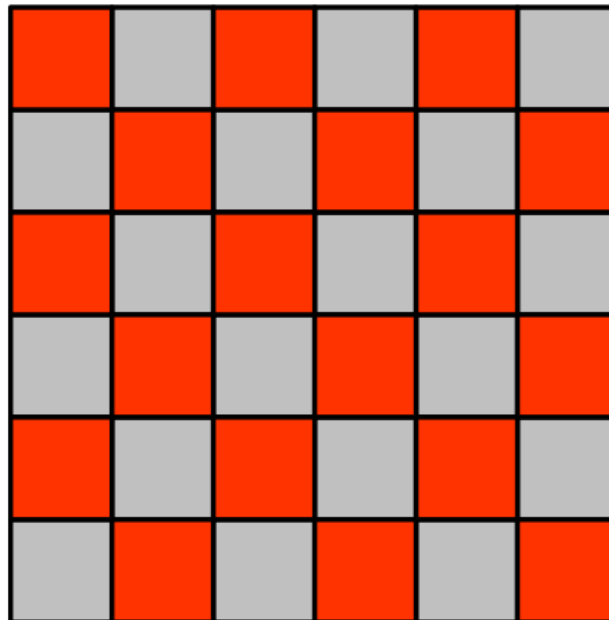
- **OpenMP is:**
 - ◆ **A great way to write fast executing code.**
 - ◆ **Your gateway to special, painful errors.**
- **You can save yourself grief if you consider the possible danger zones as you write your OpenMP programs.**
- **Tools and/or a discipline of writing portable sequentially equivalent programs can help.**

OpenMP: Case Studies

- Bubble Sort

A Special Parallel Design Pattern

Implementing a Bubble Sort in parallel is an example of a special design pattern called ***Even-Odd***, or ***Red-Black***



Non-threaded Bubble Sort

NUMN = 6

```
#include <algorithm>
...
for( int i = 0; i < NUMN; i++ )
{
    bool stop = true;

    for( int j = 0; j < NUMN-1; j++ )
    {
        if( B[ j ] > B[ j+1 ] )
        {
            std::swap( B[ j ], B[ j+1 ] );
            stop = false;
        }
    }

    if( stop )
        break;
}
```

	Step #				
original	0	1	2	3	4
6	5	4	3	2	1
5	4	3	2	1	2
4	3	2	1	3	3
3	2	1	4	4	4
2	1	5	5	5	5
1	6	6	6	6	6

Threaded Bubble Sort

```
#include <algorithm>
...
for( int i = 0; i < NUMN; i++ )
{
    int first = i % 2;    // 0 if i is 0, 2, 4, ...
                        // 1 if i is 1, 3, 5, ...

    #pragma omp parallel for default(none),shared(A,first)

    for( int j = first; j < NUMN-1; j += 2 )
    {
        if( A[ j ] > A[ j+1 ] )
        {
            std::swap( A[ j ], A[ j+1 ] );
        }
    }
}
```

NUMN = 6

	Step #					
original	0	1	2	3	4	5
6	(5	5	(3	3	(1	1
5	(6	(3	(5	(1	(3	(2
4	(3	(6	(1	(5	(2	(3
3	(4	(1	(6	(2	(5	(4
2	(1	(4	(2	(6	(4	(5
1	(2	2	(4	4	(6	6

A Comparison

NUMN = 6

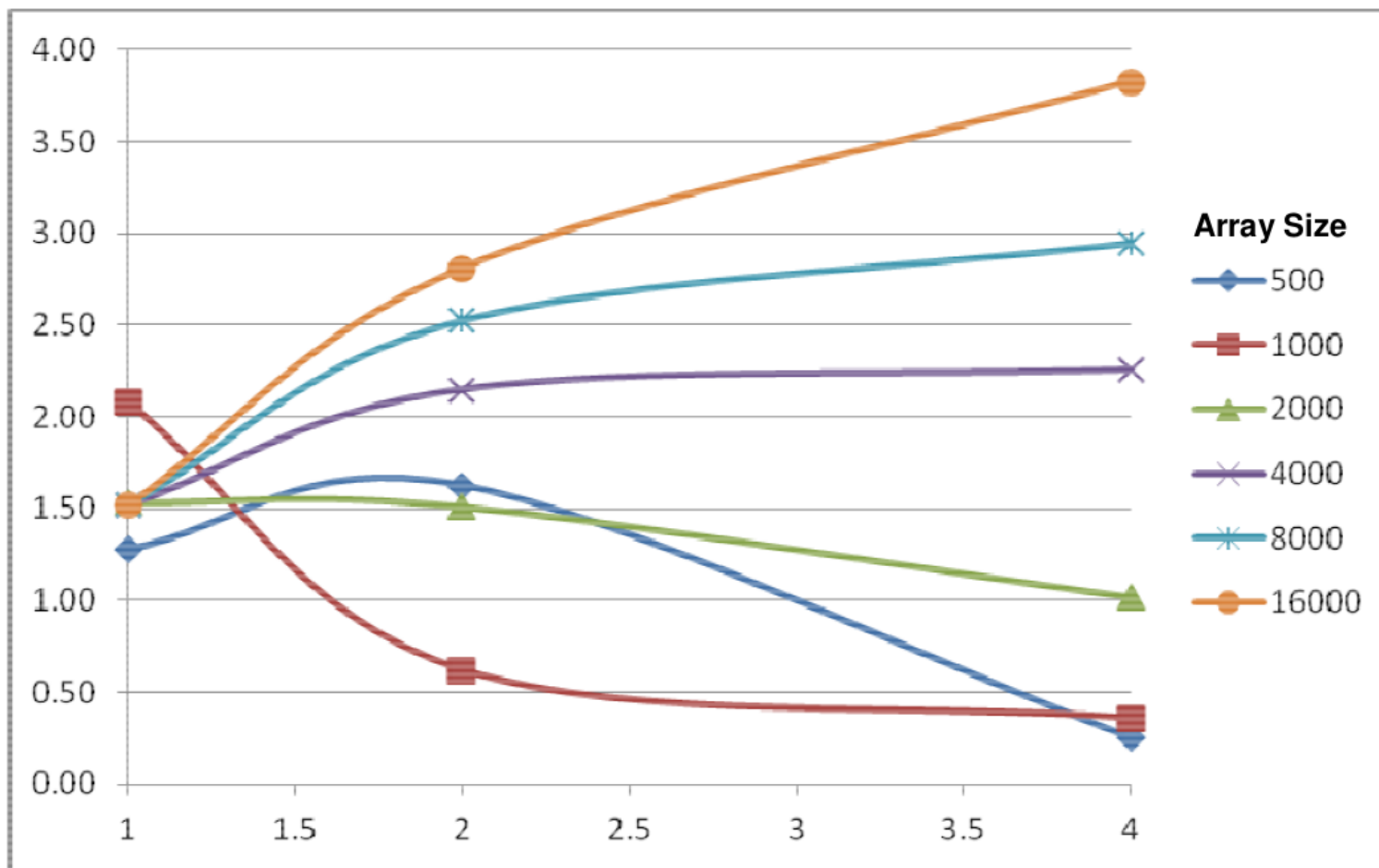
original	Step #				
	0	1	2	3	4
6	5	4	3	2	1
5	4	3	2	1	2
4	3	2	1	3	3
3	2	1	4	4	4
2	1	5	5	5	5
1	6	6	6	6	6

Non-threaded

original	Step #					
	0	1	2	3	4	5
6	5	5	3	3	1	1
5	6	3	5	1	3	2
4	3	6	1	5	2	3
3	4	1	6	2	5	4
2	1	4	2	6	4	5
1	2	2	4	4	6	6

Threaded

Speedup as a Function of # of Threads

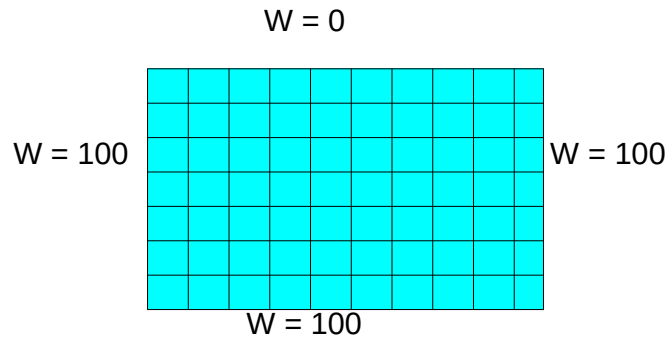


OpenMP: Case Studies

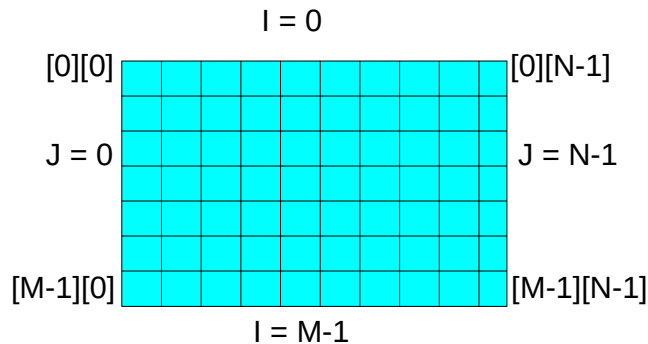
- Heated Plate

Heated plate

- This code solves the steady state heat equation on a rectangular region.
- The sequential version of this program needs approximately $18/\epsilon$ iterations to complete.
- The physical region, and the boundary conditions, are suggested by this diagram;



- The region is covered with a grid of M by N nodes, and an M by N array W is used to record the temperature.
- The correspondence between array indices and locations in the region is suggested by giving the indices of the four corners:



The steady state solution to the discrete heat equation satisfies the following condition at an interior grid point:

$$W[\text{Central}] = (1/4) * (W[\text{North}] + W[\text{South}] + W[\text{East}] + W[\text{West}])$$

where "Central" is the index of the grid point, "North" is the index of its immediate neighbor to the "north", and so on.

Given an approximate solution of the steady state heat equation, a "better" solution is given by replacing each interior point by the average of its 4 neighbors - in other words, by using the condition as an ASSIGNMENT statement:

$$W[\text{Central}] \leq (1/4) * (W[\text{North}] + W[\text{South}] + W[\text{East}] + W[\text{West}])$$

If this process is repeated often enough, the difference between successive estimates of the solution will go to zero.

This program carries out such an iteration, using a tolerance specified by the user, and writes the final estimate of the solution to a file that can be used for graphic processing.

OpenMP version

```
int main ( int argc, char *argv[] )
{
# define M 500
# define N 500

double diff;
double epsilon = 0.001;
int i, iterations, iterations_print, j;
double mean, my_diff, u[M][N], w[M][N], wtime;
/*
Set the boundary values, which don't change.
*/
mean = 0.0;

#pragma omp parallel shared ( w ) private ( i, j )
{
#pragma omp for
for ( i = 1; i < M - 1; i++ )
{
w[i][0] = 100.0;
}
```

/home/syam/ces745/openmp/others/heated_plate_openmp.c

```
#pragma omp for
```

```
for ( i = 1; i < M - 1; i++ )
```

```
    w[i][N-1] = 100.0;
```

```
#pragma omp for
```

```
for ( j = 0; j < N; j++ )
```

```
    w[M-1][j] = 100.0;
```

```
#pragma omp for
```

```
for ( j = 0; j < N; j++ )
```

```
    w[0][j] = 0.0;
```

```
/*
```

Average the boundary values, to come up with a reasonable
initial value for the interior.

```
*/
```

```
#pragma omp for reduction ( + : mean )
```

```
for ( i = 1; i < M - 1; i++ )
```

```
    mean = mean + w[i][0] + w[i][N-1];
```

```
#pragma omp for reduction ( + : mean )
```

```
for ( j = 0; j < N; j++ )
```

```
    mean = mean + w[M-1][j] + w[0][j];
```

```
}
```

/* OpenMP note:

You cannot normalize MEAN inside the parallel region. It only gets its correct value once you leave the parallel region. So we interrupt the parallel region, set MEAN, and go back in.

*/

mean = mean / (double) (2 * M + 2 * N - 4);

/* Initialize the interior solution to the mean value. */

#pragma omp parallel shared (mean, w) private (i, j)

{

#pragma omp for

for (i = 1; i < M - 1; i++)

{

for (j = 1; j < N - 1; j++)

{

w[i][j] = mean;

}

}

}

/* iterate until the new solution W differs from the old solution U by no more than EPSILON.

*/

iterations = 0;

```
diff = epsilon;
```

```
while ( epsilon <= diff )
```

```
{  
# pragma omp parallel shared ( u, w ) private ( i, j )
```

```
{
```

```
/* Save the old solution in U. */
```

```
# pragma omp for
```

```
for ( i = 0; i < M; i++ )
```

```
{
```

```
for ( j = 0; j < N; j++ )
```

```
{
```

```
u[i][j] = w[i][j];
```

```
}
```

```
}
```

```
/* Determine the new estimate of the solution at the interior points.
```

```
The new solution W is the average of north, south, east and west neighbors.
```

```
*/
```

```
# pragma omp for
```

```
for ( i = 1; i < M - 1; i++ )
```

```
{
```

```
for ( j = 1; j < N - 1; j++ )
```

```
{
```

```
w[i][j] = ( u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] ) / 4.0;
```

```
}
```

```
}
```

```
}
```

/* C and C++ cannot compute a maximum as a reduction operation.
Therefore, we define a private variable MY_DIFF for each thread.
Once they have all computed their values, we use a CRITICAL section
to update DIFF. */

```
diff = 0.0;
```

```
# pragma omp parallel shared ( diff, u, w ) private ( i, j, my_diff )  
{
```

```
    my_diff = 0.0;
```

```
# pragma omp for
```

```
    for ( i = 1; i < M - 1; i++ )
```

```
        for ( j = 1; j < N - 1; j++ )
```

```
            if ( my_diff < fabs ( w[i][j] - u[i][j] ) )
```

```
                my_diff = fabs ( w[i][j] - u[i][j] );
```

```
# pragma omp critical
```

```
    if ( diff < my_diff )
```

```
        diff = my_diff;
```

```
}
```

```
iterations++;
```

```
} // End of while loop
```

```
} // End of main()
```



Timings (on graham)

- Serial: 9.53 s.
- N=2: 4.74 s (100% efficiency).
- N=4: 2.55 s (93%).
- N=8: 1.56 s (76%).
- N=16: 1.09 s (55%).
- N=32: 1.17 s (25%).

OpenMP: Case Studies

- Mandelbrot Image

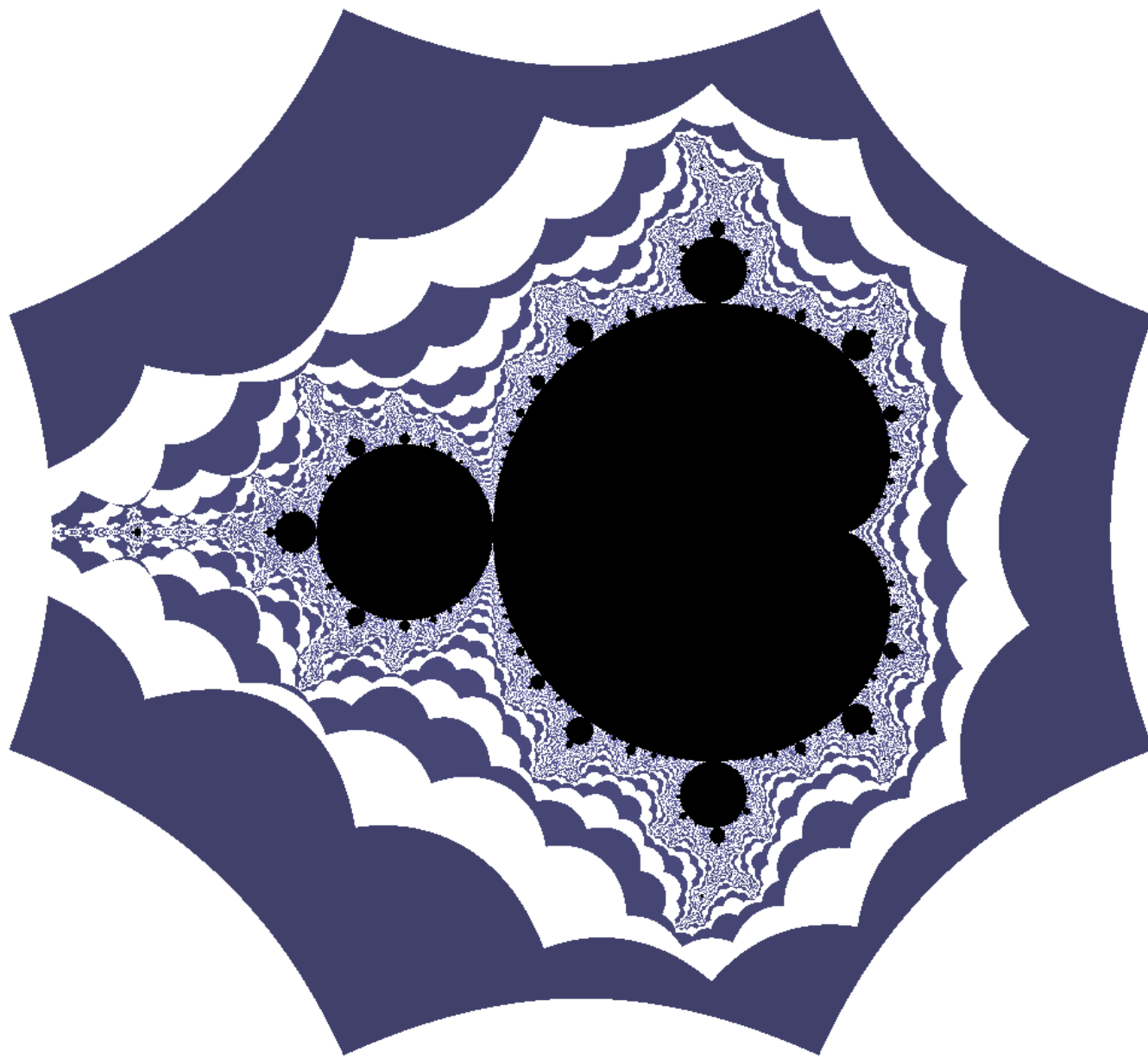
Mandelbrot Image

- MANDELBROT_OPENMP is a C program which generates an ASCII Portable Pixel Map (PPM) image of the Mandelbrot set, using OpenMP for parallel execution.
- The Mandelbrot set is a set of points C in the complex plane with the property that the iteration

$$z(n+1) = z(n)^2 + c$$

remains bounded.

- All the points in the Mandelbrot set are known to lie within the circle of radius 2 and center at the origin.
- To make a plot of the Mandelbrot set, one starts with a given point C and carries out the iteration for a fixed number of steps. If the iterates never exceed 2 in magnitude, the point C is taken to be a member of the Mandelbrot set.
- Creating an image of the Mandelbrot set requires determining the behavior of many points C under the Mandelbrot mapping. But each point can be studied independently of the others, which makes this calculation suitable for a parallel implementation.



OpenMP version

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

```
{
```

```
    int m = 1024;
```

```
    int n = 1024;
```

```
    int b[m][n], c, c_max, count[m][n], count_max = 2000, g[m][n], i, ierror, j, jhi, jlo, k, r[m][n];
```

```
    char *output_filename = "mandelbrot.ppm";
```

```
    FILE *output_unit;
```

```
    double wtime, wtime_total, x, x1, x2, y, y2, y2;
```

```
    double x_max =  1.25;
```

```
    double x_min = - 2.25;
```

```
    double y_max =  1.75;
```

```
    double y_min = - 1.75;
```

```
/*
```

```
    Carry out the iteration for each pixel, determining COUNT.
```

```
*/
```

```
# pragma omp parallel \
```

```
    shared ( b, count, count_max, g, r, x_max, x_min, y_max, y_min ) \
```

```
    private ( i, j, k, x, x1, x2, y, y1, y2 )
```

```
{
```

```
# pragma omp for
```

```
    for ( i = 0; i < m; i++ )
```

```
    {
```

```
        for ( j = 0; j < n; j++ )
```

```
        {
```

/home/syam/ces745/openmp/others/mandelbrot_openmp.c

```
x = ( ( double ) ( j - 1 ) * x_max  
      + ( double ) ( m - j ) * x_min )  
      / ( double ) ( m - 1 );
```

```
y = ( ( double ) ( i - 1 ) * y_max  
      + ( double ) ( n - i ) * y_min )  
      / ( double ) ( n - 1 );
```

```
count[i][j] = 0;
```

```
x1 = x;
```

```
y1 = y;
```

```
for ( k = 1; k <= count_max; k++ )
```

```
{  
    x2 = x1 * x1 - y1 * y1 + x;  
    y2 = 2 * x1 * y1 + y;
```

```
    if ( x2 < -2.0 || 2.0 < x2 || y2 < -2.0 || 2.0 < y2 )
```

```
    {  
        count[i][j] = k;  
        break;
```

```
    }  
    x1 = x2;  
    y1 = y2;
```

```
}
```

```

if ( ( count[i][j] % 2 ) == 1 )
{
    r[i][j] = 255;
    g[i][j] = 255;
    b[i][j] = 255;
}
else
{
    c = ( int ) ( 255.0 * sqrt ( sqrt ( sqrt (
        ( ( double ) ( count[i][j] ) / ( double ) ( count_max ) ) ) ) ) );
    r[i][j] = 3 * c / 5;
    g[i][j] = 3 * c / 5;
    b[i][j] = c;
}
}
}
}
}

```

/*

Write data to an ASCII PPM file.

*/

```

return 0;
}

```

Timings (on graham)

- Serial: 1.514 s.
- N=2: 1.008 s (75% efficiency).
- N=4: 0.992 s (38%).
- N=8: 0.877 s (22%).
- N=16: 0.625 s (15%).
- N=32: 0.507 s (9%).

OpenMP: Case Studies

- Square matrix multiplication

Square matrix multiplication

- `openmp_mmult.c` performs square matrix multiplication using OpenMP. The program generates random matrices of the dimension specified by the user and performs multiplication using a simple three-loop algorithm. Parallelism is achieved by dividing the first matrix into a group of rows for each thread for multiplication. Each thread then multiplies their group of rows with the entire second matrix, calculating a group of rows of the resultant matrix. Since the threads calculate different portions of the matrix, there was not any need to use locks or other mechanisms to protect shared memory.
- The program takes two arguments:
 - (1) The first is the number of threads to be used. If no arguments are provided, the program assumes 4 threads.
 - (2) The second is the dimension of the matrices to be multiplied. If only the first argument is provided, the program uses 100x100 matrices.
- The program prints the time required for multiplication and writes the three matrices to the files 'A.txt', 'B.txt', and 'C.txt'.

The code

```
int main(int argc, char *argv[] ) {
    double *mat1=NULL, *mat2=NULL, *mat3=NULL; //matrices to be multiplied, and the result
    int nthr; //number of threads
    int dim; //matrix dimension

    nthr = 4; // the number of threads
    dim = 1000; // the number of rows of the first matrix

    //allocate memory for the three matrices
    mat1 = (double *)malloc(dim*dim*sizeof(double));
    mat2 = (double *)malloc(dim*dim*sizeof(double));
    mat3 = (double *)malloc(dim*dim*sizeof(double));

    //get the two matrices to be multiplied
    srand( (unsigned int)time(NULL) );
    rand_mat(mat1, dim);
    rand_mat(mat2, dim);

    mat_mult_thr(mat1,mat2,mat3,dim,nthr);

    //free all of the memory
    free(mat1);
    free(mat2);
    free(mat3);

    return 0;}
```

/home/syam/ces745/openmp/others/openmp_mmult.c


```

//mat_mult_thr() does parallel (threaded) matrix multiplication
// mat1 and mat2 are the matrices to be multiplied
// result is the result matrix
// dim is the number of rows/columns of each matrix
// nthr is the number of threads
void mat_mult_thr(double *mat1, double *mat2, double *result, int dim, int nthr)
{
    int part_rows, th_id;
    part_rows = dim/nthr;

    omp_set_num_threads(nthr); //set the number of threads
    #pragma omp parallel shared(mat1,mat2,result,part_rows)
    {
        int i,j,k; //iterators

        //Split the first for loop among the threads
        #pragma omp for schedule(guided,part_rows)
        for(i=0; i<dim; i++) //iterate through the rows of the result
        {
            for(j=0; j<dim; j++) //iterate through the columns of the result
            {
                *( result+(j+i*dim) ) = 0; //initialize

                //iterate through the inner dimension (columns of first matrix/rows of second matrix)
                for(k=0; k<dim; k++)
                    *( result+(j+i*dim) ) += *( mat1+(k+i*dim) )*( *( mat2+(j+k*dim) ));
            }
        }
    }
}

```

Timings for 2000² size (on graham)

- Serial: 15.12 s.
- N=2: 7.66 s (99% efficiency).
- N=4: 3.94 s (96%).
- N=8: 2.08 s (91%).
- N=16: 1.14 s (83%).
- N=32: 0.686 s (69%).

End