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

```
C$OMP PARALLEL SECTIONS
      A = B + C
C$OMP SECTION
      B = A + C
C$OMP SECTION
      C = B + A
C$OMP END PARALLEL SECTIONS
```

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

Race Conditions:
A complicated solution

```
      ICOUNT = 0
C$OMP PARALLEL SECTIONS
      A = B + C
      ICOUNT = 1
C$OMP FLUSH ICOUNT
C$OMP SECTION
1000 CONTINUE
C$OMP FLUSH ICOUNT
      IF(ICOUNT .LT. 1) GO TO 1000
      B = A + C
      ICOUNT = 2
C$OMP FLUSH ICOUNT
C$OMP SECTION
2000  CONTINUE
C$OMP FLUSH ICOUNT
       IF(ICOUNT .LT. 2) GO TO 2000
      C = B + A
C$OMP END PARALLEL SECTIONS
```

- 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

```
c$omp parallel shared(x) private(tmp)
        id = omp_get_thread_num()
c$omp do reduction(+:x)
        do j=1,100
            tmp = work(j)
            x = x + tmp
        enddo
c$omp end do nowait
        y(id) = work(x,id)
c$omp end parallel
```

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

Examples: Race Conditions

```
  real :: tmp,x
c$omp parallel do reduction(+:x)
  do j=1,100
    tmp = work(j)
    x = x + tmp
  enddo
c$omp end do
  y(id) = 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 <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{              int i;       double  x, sum, pi=0.0;
               step = 1.0/(double) num_steps;
               omp_set_num_threads(NUM_THREADS)
#pragma omp parallel private (x, sum)
{
          int  id = omp_get_thread_num();
               for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
                         x = (i+0.5)*step;
                         sum += 4.0/(1.0+x*x);
               }
#pragma omp  critical
               pi += sum
}
}
```

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

```
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.139293
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.139667
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.139493
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.140276
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.137184
syam@orc131:[~/ces745/openmp/C/pi/DEMO] ./pi-critical2
PI = 3.140468
```

Data Dependences

Removal

A loop containing multiple data dependencies

```
    do i = 2, n-1
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
    enddo
```

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 is assigned before start of loop
    do i = 1, n-1
        x = (b(i) + c(i))/2
1       a(i) = a(i+1) +x
    enddo
```

Parallel version with dependencies removed

```
! $omp parallel do shared(a, a2)
    do i = 1, n-1
       a2(i) = a(i+1)                    - make a copy of the array
    enddo
! $omp parallel do shared(a, a2) private(x)
    do i = 1, n-1
        x = (b(i) + c(i))/2
10      a(i) = a2(i) +x
    enddo
```

Remove dependencies

- **removal of output dependencies**

Serial version containing output dependencies

```
do i = 1, n
     x = (b(i) + c(i))/2
    a(i) = a(i) +x
    d(1) = 2 * x
enddo
y = x + d(1) + d(2)
```

Parallel version with dependencies removed

```
! $omp parallel do shared(a) lastprivate(x, d1)
    do i = 1, n
         x = (b(i) + c(i))/2
        a(i) = a(i) +x
        d1 = 2 * x
    enddo
    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
      do i = 1, n
         x = x + a(i)
      enddo
```

Parallel version with dependencies removed by reduction clause

```
      x = 0
! $omp parallel do reduction(+: x)
      do i = 1, n
         x = x + a(i)
      enddo
```

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

```
      do i = 2, n
 10      b(i) = b(i) + a(i-1)
 20      a(i) = a(i) + c(i)
      enddo
```

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(2) = b(2) + a(1)
! $omp parallel do shared(a, b, c)
      do i = 2, n-1
 20      a(i) = a(i) + c(i)
 10      b(i+1) = b(i+1) + a(i)
      enddo
      a(n) = a(n) + c(n)
```

Dealing with non-removable dependencies

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

Serial version containing a recurrence

```
do j = 2, n
   do i = 1, n
        a(i, j) = a(i, j) + a(i, j-1)
   enddo
enddo
```

Parallel version to the loop in the nest

```
   do j = 2, n
!$omp parallel do shared (a)
   do i = 1, n
       a(i, j) = a(i, j) + a(i, j-1)
   enddo
enddo
```

Dealing with non-removable dependencies

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

Serial version containing a recurrence

```
        do j = 2, n
10          a(j) = a(j) + a(j-1)
20          y = y + c(j)
        enddo
```

Parallel version

```
       do j = 2, n
10          a(j) = a(j) + a(j-1)
        enddo

!$omp parallel do reduction(+: y)
        do j = 2, n
 20         y = y + c(j)
        enddo
```

/home/syam/ces745/openmp/OmpSCR_v2.0/applications/c_LoopsWithDependencies

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

```
        call OMP_INIT_LOCK(lcka)
        call OMP_INIT_LOCK(lckb)
c$omp parallel sections
        call OMP_SET_LOCK(lcka)
        call OMP_SET_LOCK(lckb)
        call useAandB(res)
        call OMP_UNSET_LOCK(lckb)
        call OMP_UNSET_LOCK(lcka)
c$omp section
        call OMP_SET_LOCK(lckb)
        call OMP_SET_LOCK(lcka)
        call useBandA(res)
        call OMP_UNSET_LOCK(lcka)
        call OMP_UNSET_LOCK(lckb)
c$omp end parallel sections
```

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

Examples: Deadlock

```
        call OMP_INIT_LOCK(lcka)
c$omp parallel sections
        call OMP_SET_LOCK(lcka)
        ival = work()
        if (ival.eq.tol) then
          call OMP_UNSET_LOCK(lcka)
        else
          call error(ival)
        endif
c$omp section
        call OMP_SET_LOCK(lcka)
        call useBandA(res)
        call OMP_UNSET_LOCK(lcka)
c$omp end parallel sections
```

- 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

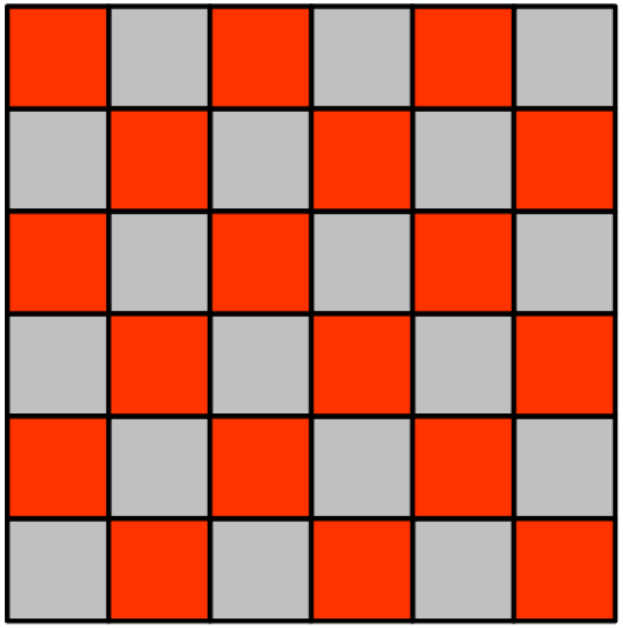
- OpenMP is:
  - A great way to write fast executing code.
  - Your gateway to special, painful errors.
- You can save yourself grief if you consider the possible danger zones as you write your OpenMP programs.
- Tools and/or a discipline of writing portable sequentially equivalent programs can help.

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***



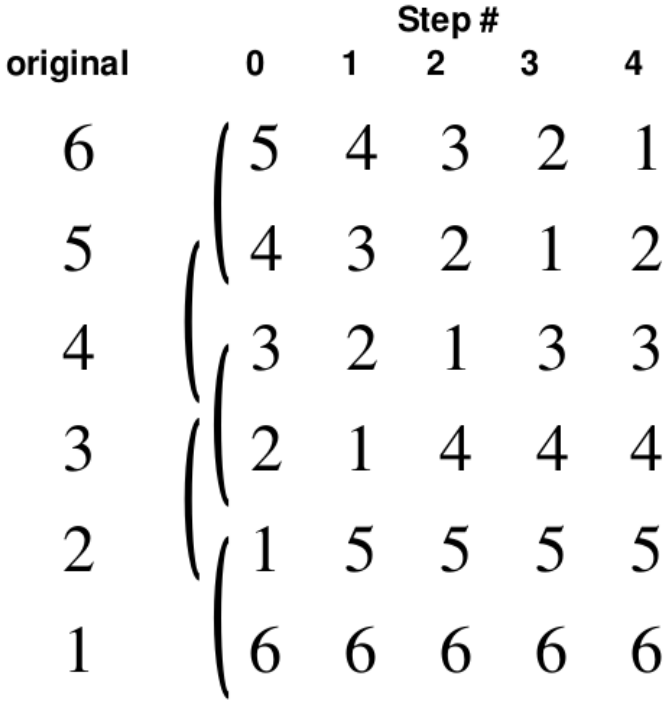
Non-threaded Bubble Sort

```
#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;
}
```

NUMN = 6



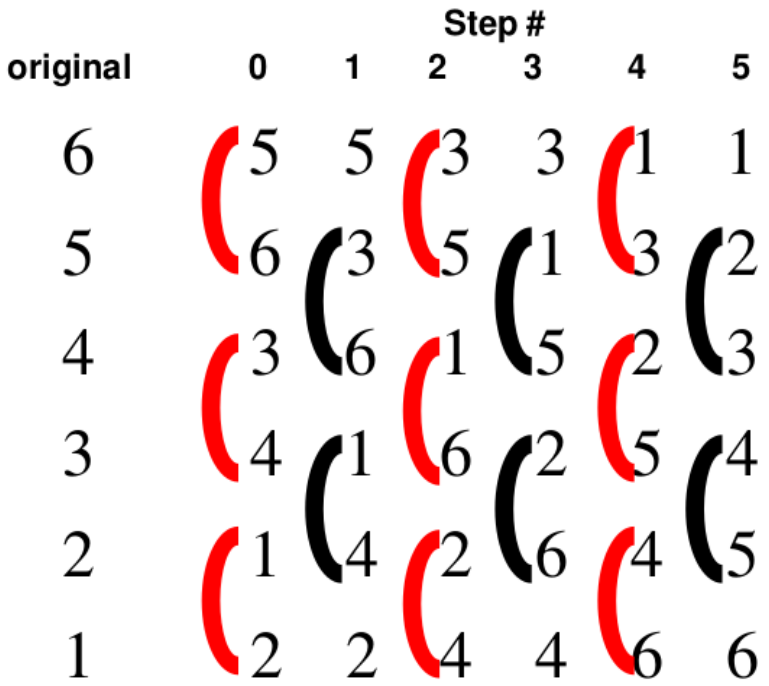
Threaded Bubble Sort

```
#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



A Comparison

NUMN = 6

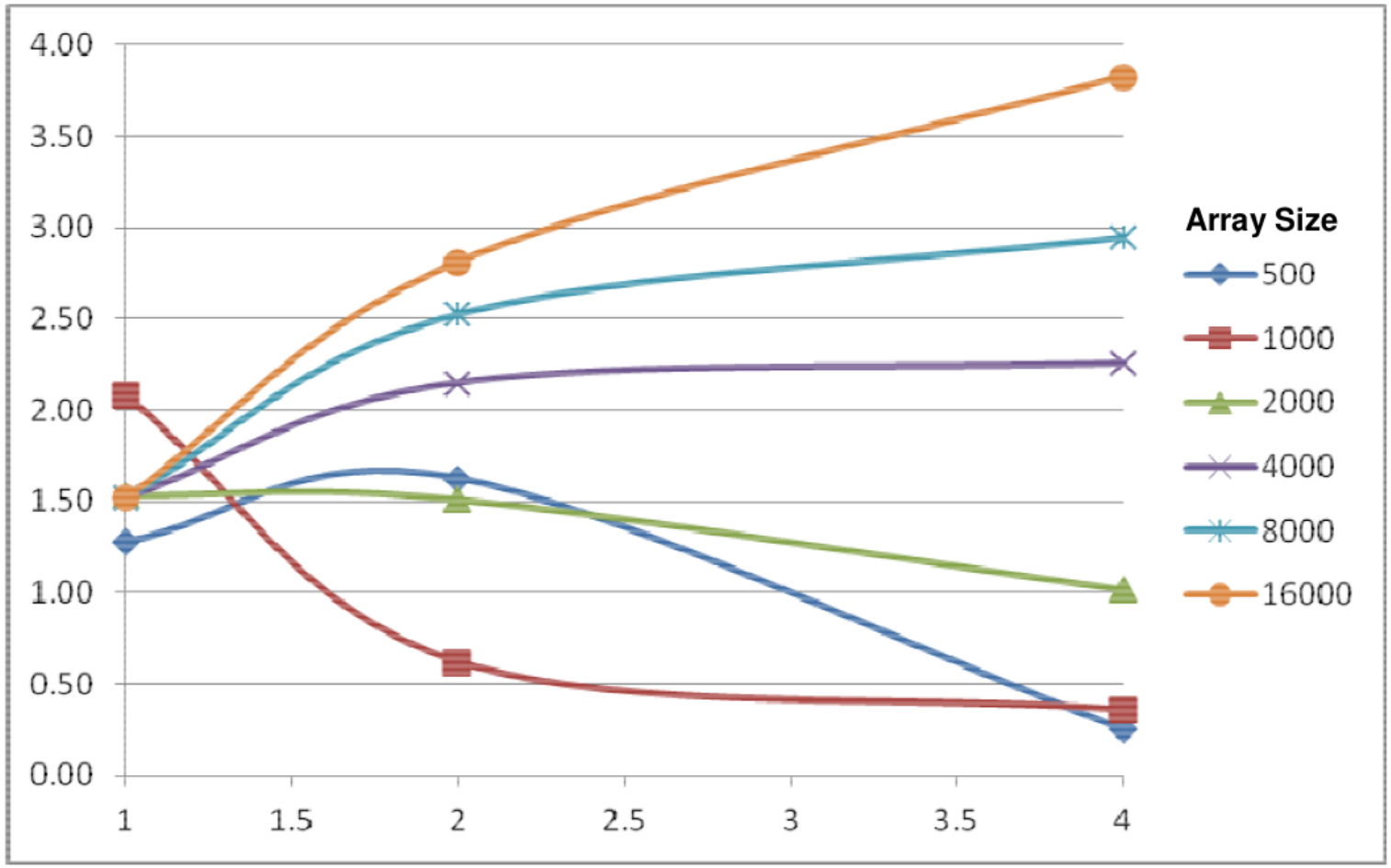
	Step #				
original	0	1	2	3	4
6	5	4	3	2	1
5	4	3	2	1	2
4	3	2	1	3	3
3	2	1	4	4	4
2	1	5	5	5	5
1	6	6	6	6	6

Non-threaded

	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

Threaded

Speedup as a Function of # of Threads

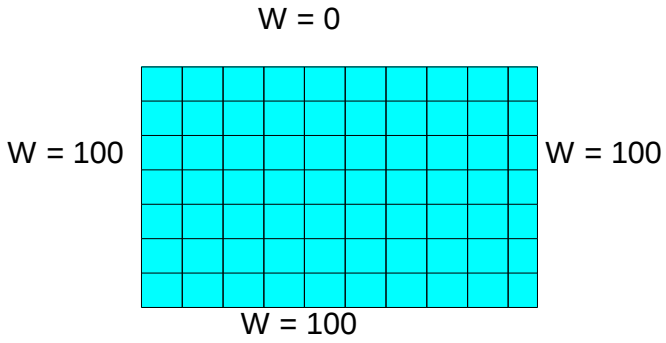


OpenMP: Case Studies

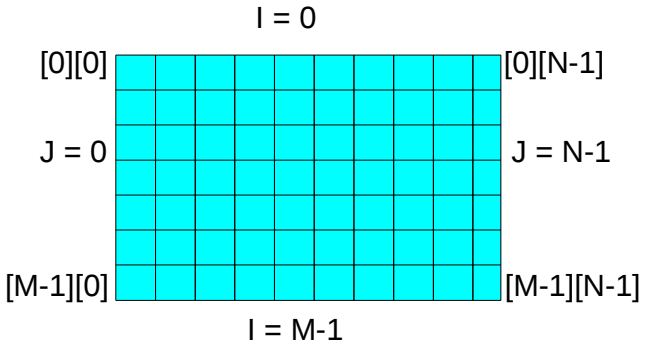
- Heated Plate

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:



http://people.sc.fsu.edu/~jburkardt/c_src/heated_plate/heated_plate.html

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

W[Central] = (1/4) * (W[North] + W[South] + W[East] + W[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[Central] <= (1/4) * (W[North] + W[South] + W[East] + W[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 orca)

- Serial: 25.9 s.
- N=2: 16.7 s (78% efficiency).
- N=4: 8.21 s (79%).
- N=8: 3.98 s (81%).
- N=16: 3.04 s (53%).
- N=24: 3.31 s (33%).

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

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

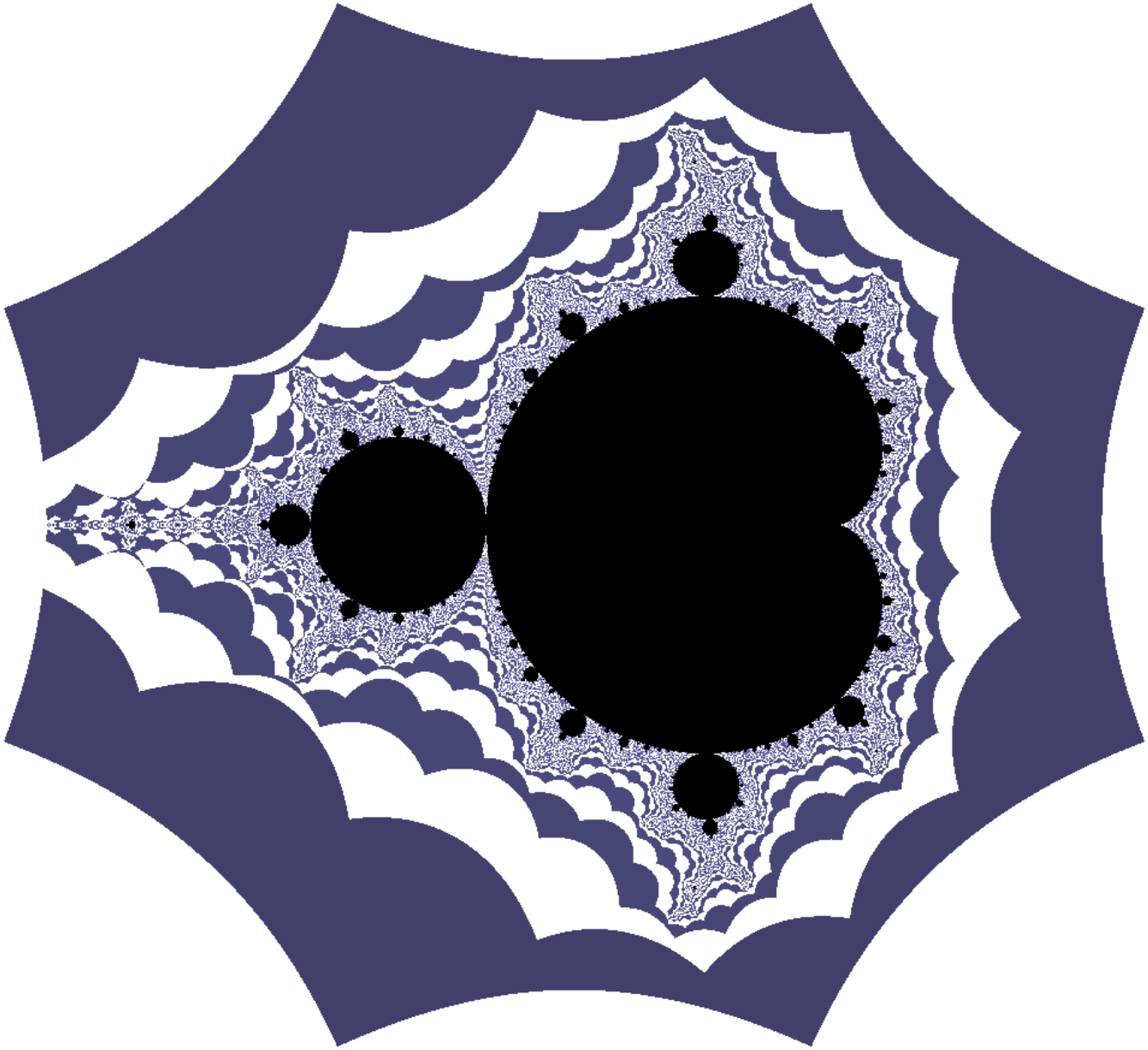
  remains bounded.

- 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.

http://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_openmp/mandelbrot_openmp.html



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 orca)

- Serial: 2.34 s.
- N=2: 1.18 s (~100% efficiency).
- N=4: 1.16 s (50%).
- N=8: 0.88 s (33%).
- N=16: 0.52 s (28%).
- N=24: 0.37 s (26%).

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'.

http://www.arc.vt.edu/resources/software/openmp/docs/openmp_mmult.c

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 (on orca)

- Serial: 8.16 s.
- N=2: 4.27 s (96% efficiency).
- N=4: 2.22 s (92%).
- N=8: 1.15 s (89%).
- N=16: 0.67 s (76%).
- N=24: 0.65 s (52%).

End