Shared Memory Parallel Programming

Introduction to OpenMP

These slides were originally written by Dr. Barbara Chapman, University of Houston

OpenMP Components

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

Parallel Regions

- You create threads in OpenMP with the "omp parallel" pragma.
- For example, To create a 4 thread parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

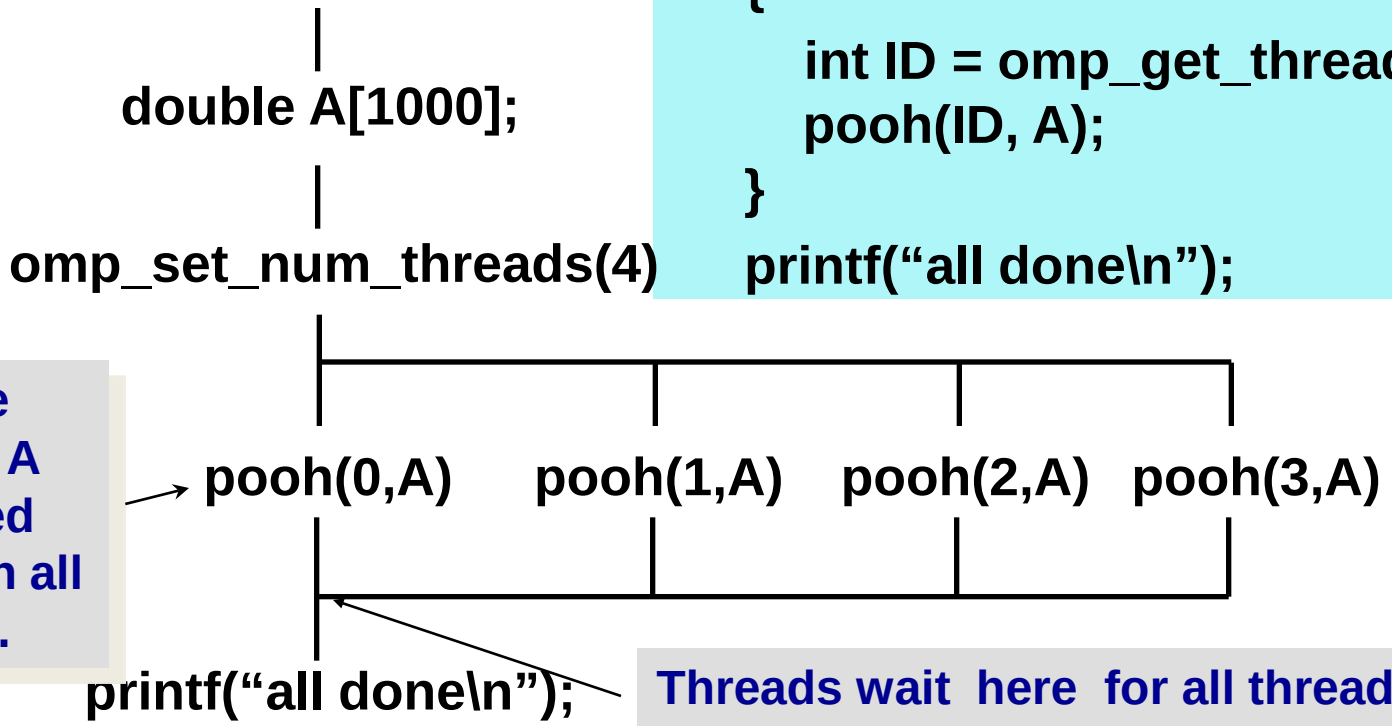
Runtime function returning a thread ID

- **Each thread calls pooh(ID,A) for `ID = 0` to 3**

Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

Exercise:

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
void main()
{
    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{
#pragma omp parallel
 {
    int ID =
omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  } }
```

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(2)

world(3)

Parallel Regions and the “if” clause

Active vs *inactive* parallel regions.

- An optional if clause causes the parallel region to be active only if the logical expression within the clause evaluates to true.
- An if clause that evaluates to false causes the parallel region to be inactive (i.e. executed by a team of size one).

```
double A[N];

#pragma omp parallel if(N>1000)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Worksharing Constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for schedule(static)
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP Worksharing Constructs

- Divides the execution of the enclosed code region among the members of the team
- The "for" worksharing construct splits up loop iterations among threads in a team
  - Each thread gets one or more "chunk"

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++) {
      work(i);
}
```

By default, there is a barrier at the end of the "omp for". Use the "nowait" clause to turn off the barrier.

#pragma omp for nowait

"nowait" is useful between two consecutive, independent omp for loops.

omp do in Fortran

Loop Collapse

- Allows parallelization of perfectly nested loops without u
- The colla dicates how many loo

```
NESTED PARALLELISM
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return(0);
}
```

```
!$omp end parallel do
```

OpenMP schedule Clause

schedule (static | dynamic | guided [, chunk])
schedule (auto | runtime)

static	Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
dynamic	Fixed portions of work; size is controlled by the value of chunk; When a thread finishes, it starts on the next portion of work
guided	Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially
auto	The compiler (or runtime system) decides what is best to use; choice could be implementation dependent
runtime	Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE

The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

20 iterations

6 threads

Static schedule

3 iterations per thread → last thread has 5 iterations

4 iterations per thread → last thread has 0 iterations !

OpenMP Sections

- Worksharing construct
- Gives a different structured block to each thread

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
        x_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

NOWAIT Clause

Whenever … work
faster tha… their
respective

#pragi

“nowai
consec
loops.

```
#pragma omp parallel
{
  #pragma omp for
  for(int n=0; n<10; ++n) Work();

  // This line is not reached before the for-loop is completely finis
  SomeMoreCode();
}

// This line is reached only after all threads
// the previous parallel block are finished.
CodeContinues();

#pragma omp parallel
{
  #pragma omp for nowait
  for(int n=0; n<10; ++n) Work();

  // This line may be reached while some threads are … executing the for-loop.
  SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();
```

The nowait directive can only be attached to sections, for and single.

OpenMP Master

- Denotes a structured block executed by the master thread
- The other threads just skip it
  - no synchronization is implied

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma barrier
        do_many_other_things();
}
```

OpenMP Single

- Denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

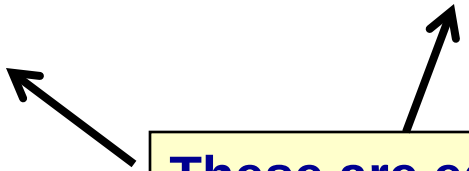
```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

Combined parallel/work-share

- OpenMP shortcut: Put the “parallel” and the work-share on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```



- **There’s also a “parallel sections” construct.**

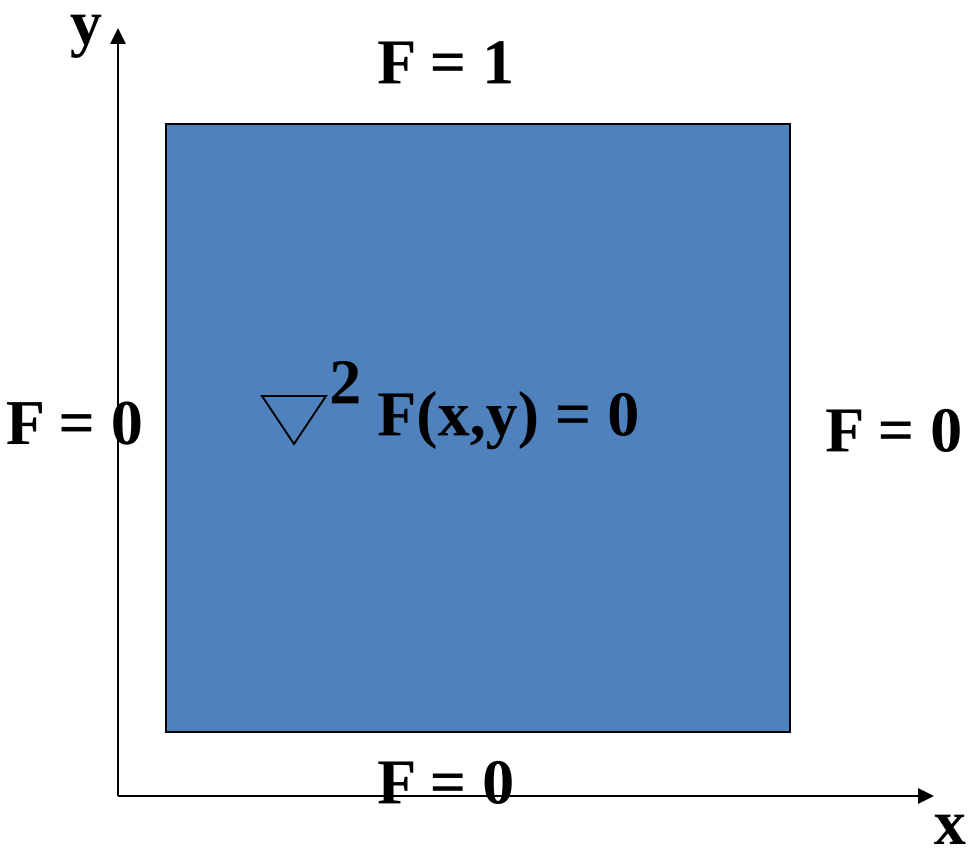
Laplace Equation

- An elliptic partial differential equation (PDE)
- Can model many natural phenomena, e.g., heat dissipation in a metal sheet
- PDEs used to model many physical systems (weather, flow over wing, turbulence, etc.)

Laplace Equation

- Typical approach is to generate mesh by covering region of interest with a grid of points
- Then impose an initial state or initial approximate solution on grid
- At each time step, update current values at each point on the grid
- Terminate either after a specified number of iterations, or when a steady state is reached

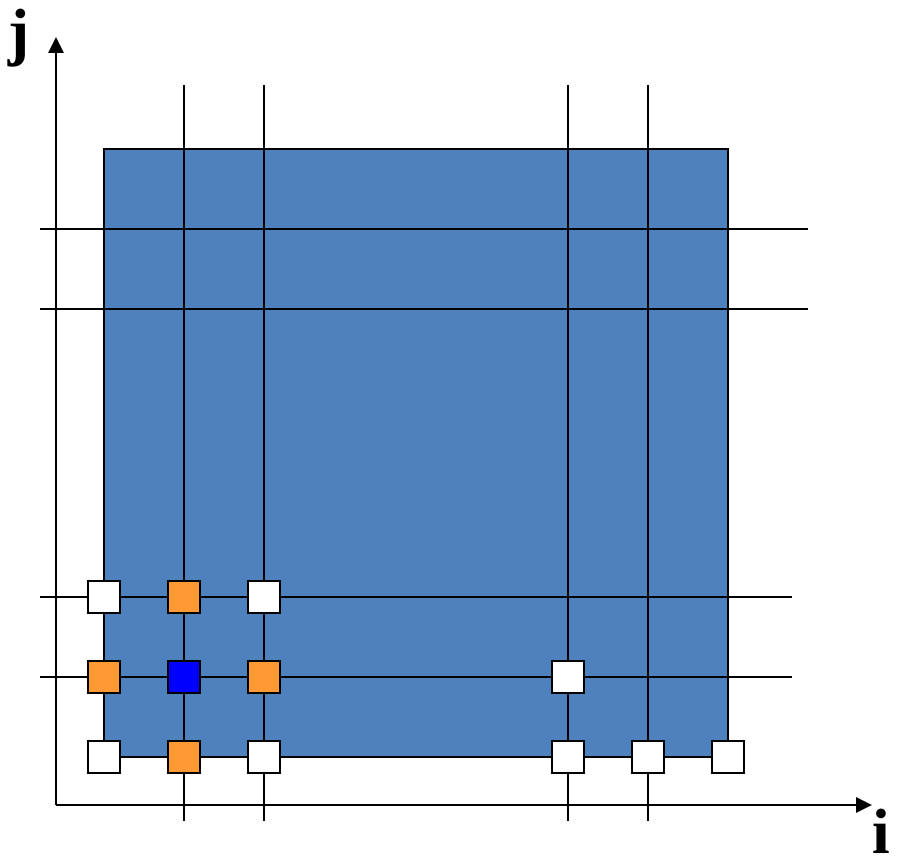
Problem Statement



Discretization

- Represent F in continuous rectangle by a 2-dimensional discrete grid (array)
- The boundary conditions on the rectangle are the boundary values of the array
- The internal values are found by updating values using a combination of values at neighboring grid points until termination

Discretized Problem Statement



4-point stencil

Solution Methods

- At each time step, update solution and test for steady state
- Variety of methods for update operation
  - Jacobi, Gauss-Seidel, SOR (Successive Over-Relaxation), Red-Black. Multigrid,…
- Test for steady state by comparing values at grid points from previous time step with those at current time step

Typical Algorithm

- For some number of iterations

  for each internal grid point

  update value using average of its neighbors

- Termination condition:

  values at grid points change very little between one iteration and the next

  (we will ignore this part in our example)

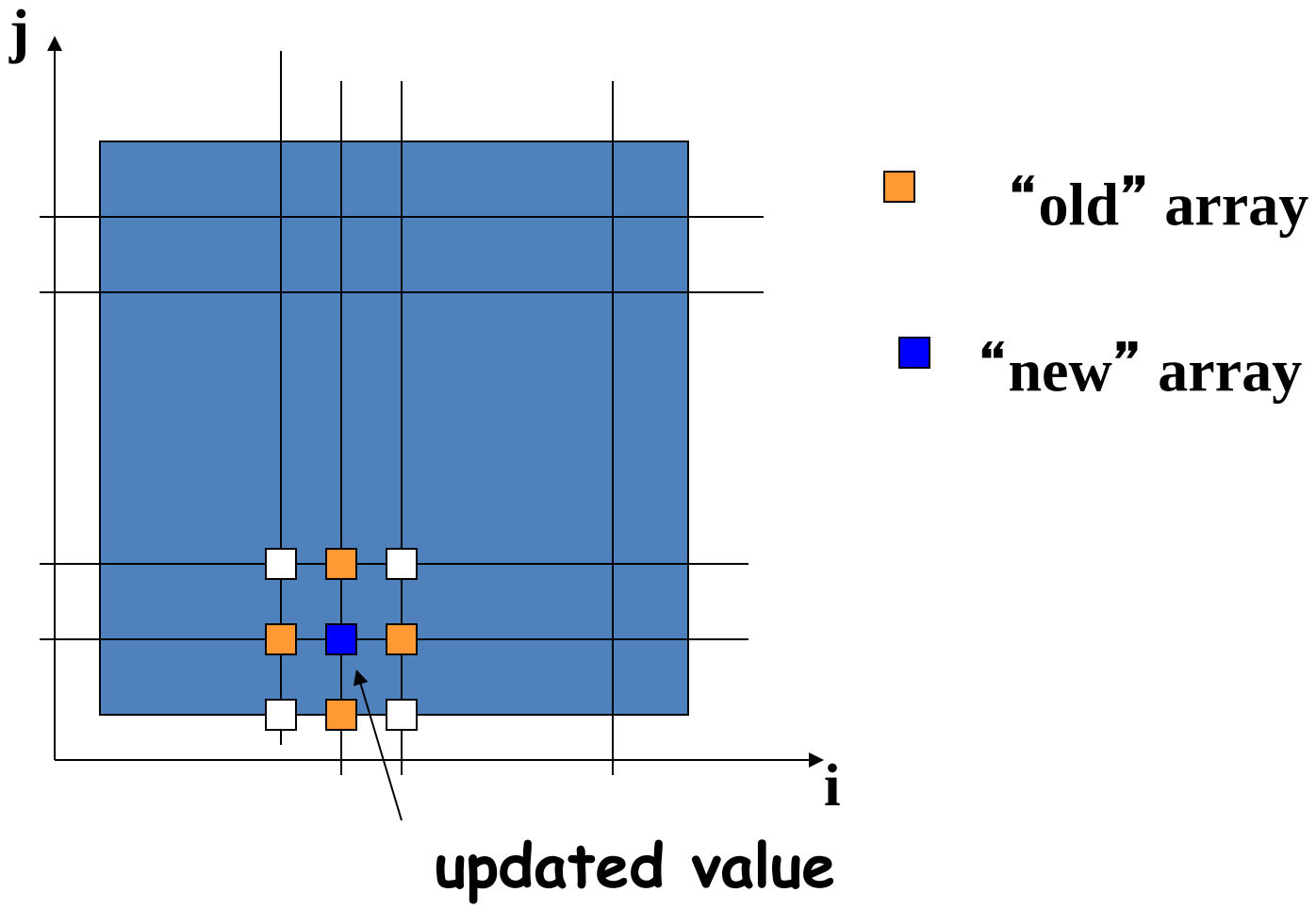
Jacobi Method

```
/* Initialization */
for( i=0; i<n+1; i++ ) grid[i][0] = 0.0;
for( i=0; i<n+1; i++ ) grid[i][n+1] = 0.0;
for( j=0; j<n+1; j++ ) grid[0][j] = 1.0;
for( j=0; j<n+1; j++ ) grid[n+1][j] = 0.0;
for( i=1; i<n; i++ )
   for( j=1; j<n; j++ )
      grid[i][j] = 0.0;
```

Jacobi Method

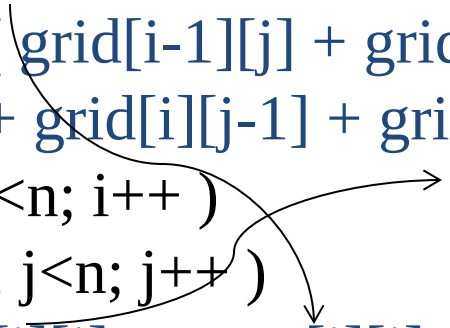
```
for some number of timesteps/iterations {
   for (i=1; i<n; i++ )
      for( j=1, j<n, j++ )
            temp[i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1] );
   for( i=1; i<n; i++ )
      for( j=1; j<n; j++ )
            grid[i][j] = temp[i][j];
}
```

Data Usage in Parallel Jacobi



Jacobi Method

```
for some number of timesteps/iterations {
  for (i=1; i<n; i++ )
    for( j=1, j<n, j++ )
        temp[i][j] = 0.25 *
            ( grid[i-1][j] + grid[i+1][j]
            + grid[i][j-1] + grid[i][j+1] );
  for( i=1; i<n; i++ )
    for( j=1; j<n; j++ )
        grid[i][j] = temp[i][j];
}
```



temp defined here and used below.

grid defined here and used above in next timestep

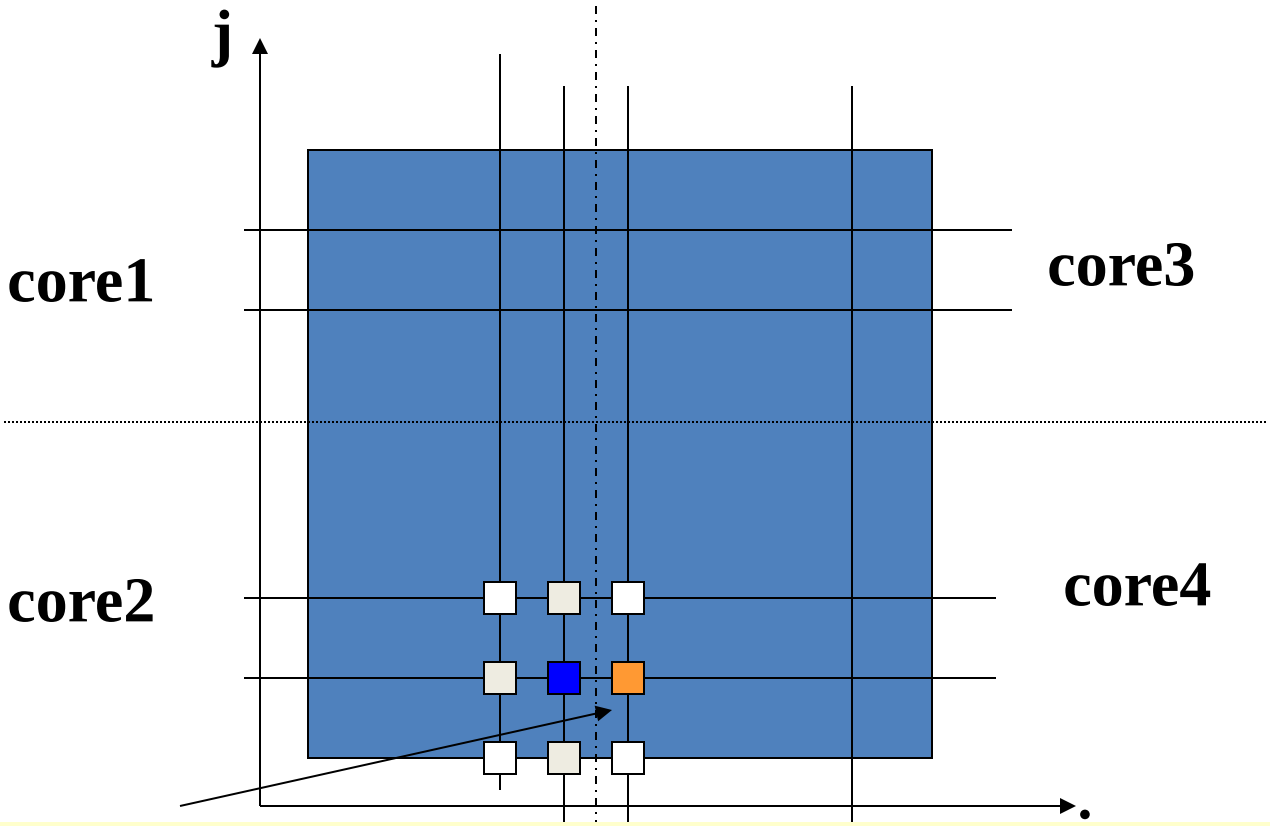
Parallel Jacobi Method

- No dependences between iterations of first (i,j) loop nest
- No dependences between iterations of second (i,j) loop nest
- True and anti-dependence between first and second loop nest in the same timestep
- True dependence between second loop nest and first loop nest of next timestep

Parallel Jacobi (continued)

- First (i,j) loop nest can be parallelized
- Second (i,j) loop nest can be parallelized
- But keep order of loops and timesteps
  - Processors / cores must not begin second loop until first loop nest completes
  - or begin new timestep until previous completes
- Threads wait for each other at the end of each (i,j) loop nest
- This is a barrier (barrier synchronization)

E.g. Data Usage in Parallel Jacobi



Updated by thread on another core. Needs to be back in main memory before next loop.

Parallel Jacobi Method

```
for some number of timesteps/iterations {
   for (i=1; i<n; i++ )          ← parallel, distribute iterations
      for( j=1, j<n, j++ )
            temp[i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  grid[i][j-1] + grid[i][j+1] );
… synchronization point …
   for( i=1; i<n; i++ )          ← parallel, distribute iterations
      for( j=1; j<n; j++ )
            grid[i][j] = temp[i][j];
… synchronization point …
}
```

OpenMP Jacobi Method

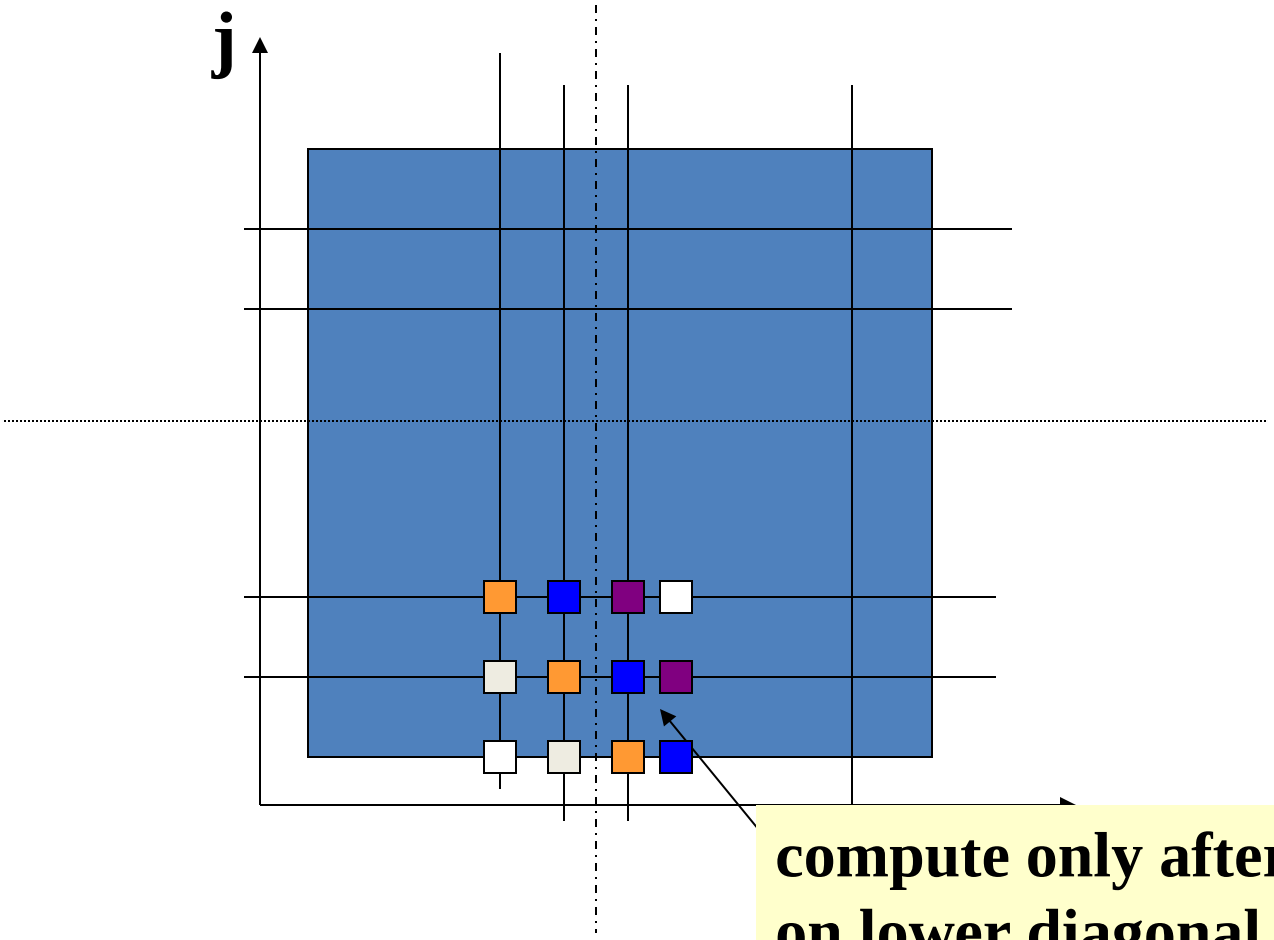
```
for some number of timesteps/iterations {
#pragma omp parallel for
for (i=1; i<n; i++ )
      for( j=1, j<n, j++ )
            temp[i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  grid[i][j-1] + grid[i][j+1] );
#pragma omp parallel for
for( i=1; i<n; i++ )
      for( j=1; j<n; j++ )
            grid[i][j] = temp[i][j];
}
```

OpenMP automatically inserts a barrier at the end of each parallel loop

Gauss-Seidel Method

```
for some number of timesteps/iterations {
   for (i=1; i<n; i++ )
      for( j=1, j<n, j++ )
            grid [i][j] = 0.25 *
                ( grid[i-1][j] + grid[i+1][j]
                + grid[i][j-1] + grid[i][j+1] );
   }
```

Data Parallel Computation

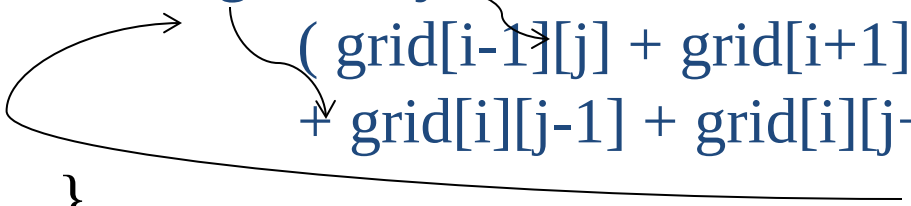


Parallel Gauss-Seidel

- Uses less memory and converges faster
- Doesn’t require second loop
- But dependences and anti-dependences in loop nest
- There is some concurrency: but it is hard to exploit here

Gauss-Seidel Method

```
for some number of timesteps/iterations {
   for (i=1; i<n; i++ )
      for( j=1, j<n, j++ )
            grid [i][j] = 0.25 *
               ( grid[i-1][j] + grid[i+1][j]
                + grid[i][j-1] + grid[i][j+1] );
   }
```



Gauss-Seidel Method

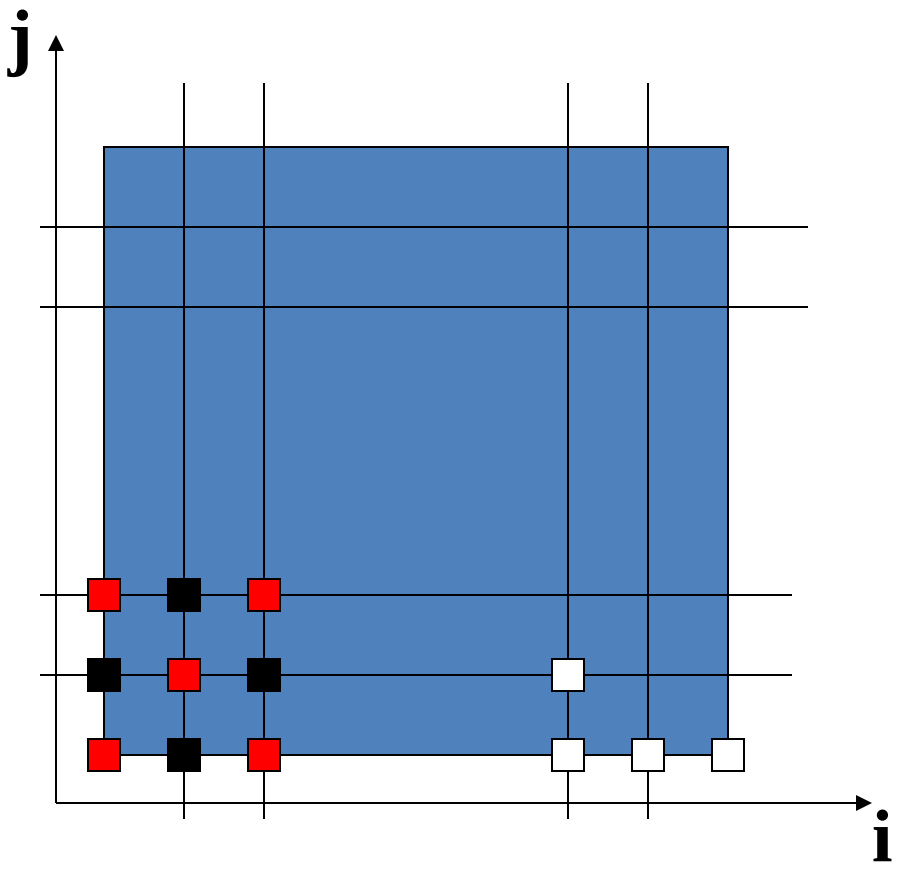
```
for some number of timesteps/iterations {
  for (i=1; i<n; i++ ) ← not parallel
    for( j=1, j<n, j++ ) ← not parallel
        grid [i][j] = 0.25 *
            ( grid[i-1][j] + grid[i+1][j]
            + grid[i][j-1] + grid[i][j+1] );
  }
```

Neither loop is parallel

Red-Black Method

- Grid points partitioned into two sets like a chess board
  - "colored" red and black
- Update in two steps
  - Compute new values on "red" points using current values on neighboring "black" points
  - Compute new values on "black" points using current values on neighboring "red" points
- Doesn't require temporary array

Red-Black Grid Points



Red-Black Method

```
for some number of timesteps/iterations {// update red points
   for (i=1; i<n; i+=2 ) ← parallel
      for( j=1, j<n, j+=2 ) ← parallel
            grid [i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1] );
for (i=2; i<n; i+=2 ) ← parallel
      for( j=2, j<n, j+=2 ) ← parallel
            grid [i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1] );
   }
```

Red-Black Method

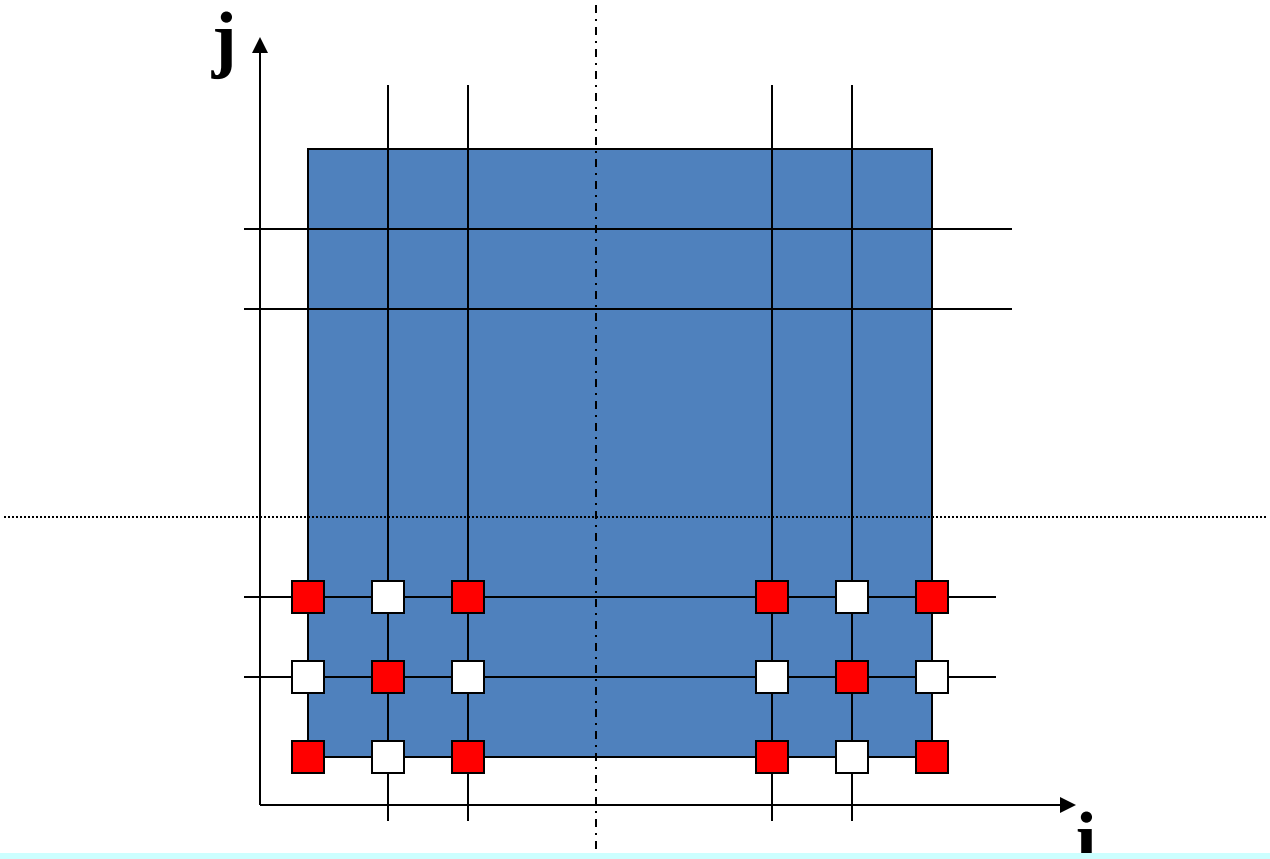
```
for some number of timesteps/iterations {// update black points
   for (i=1; i<n; i+=2 ) ← parallel
      for( j=2, j<n, j+=2 ) ← parallel
            grid [i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1] );
for (i=2; i<n; i+=2 ) ← parallel
      for( j=1, j<n, j+=2 ) ← parallel
            grid [i][j] = 0.25 *
                  ( grid[i-1][j] + grid[i+1][j]
                  + grid[i][j-1] + grid[i][j+1] );
   }
```

Parallel Red-Black

- Uses same amount of memory as Gauss-Seidel
- Converges more slowly but better than Jacobi

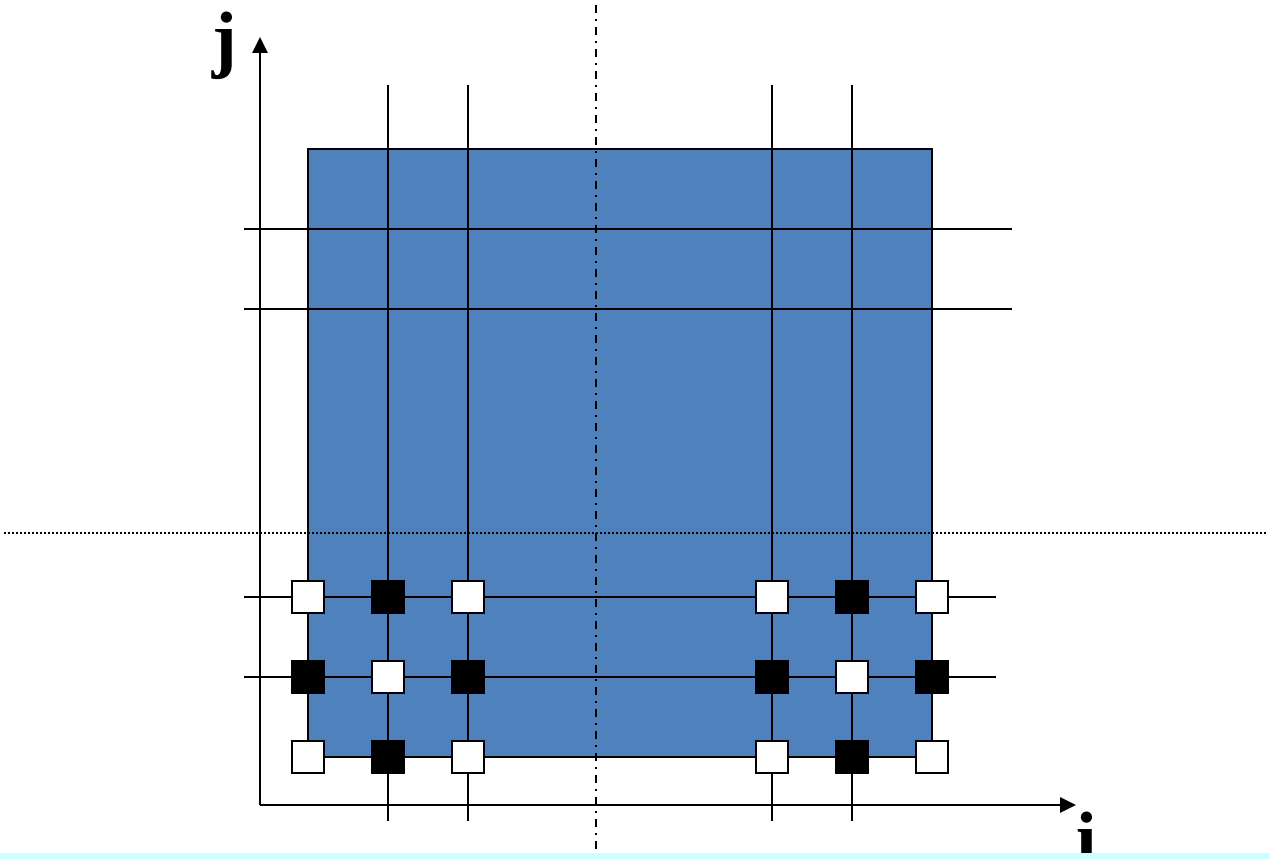
- A popular compromise for parallel computations

Red-Black Method



Update all red points in parallel

Red-Black Method



Then update all black points in parallel

Parallel Red-Black

- Essentially splits Gauss-Seidel computation into two loops (two pairs in this formulation)
- No dependences within any of loops
- But dependences between update of red points and update of black points
- So need barrier between 2nd and 3rd loop, and at end of each iteration

Reading Material

- Download the OpenMP 3.1 Application Programming Interface from www.openmp.org
- Read up on the worksharing directives for C++
- There are a number of clauses that can be used together with them
- Read the first three chapters of “Using OpenMP”