Performance Issue

Cache Behavior, Jacobi relaxation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

OpenMP Performance Issues

- Using OpenMP to parallelize a program isn't as simple as it looks
  - "Just add a few compiler directives"
  - Especially when considering performance issues
  - To get good performance requires many changes to the original program
  - Write programs in explicit parallelism form
    - I.e., "**`omp parallel`**" rather than "**`omp parallel for`**"
    - Almost no difference between OpenMP and pthreads* in terms of programmability
      - Not quite true … compiler support for reductions, privatization, and mixing of loop scheduling creates some benefits over explicit programming with pthreads

*Pthreads – an explicit programming model we will cover elsewhere in the course

OpenMP Performance Issues

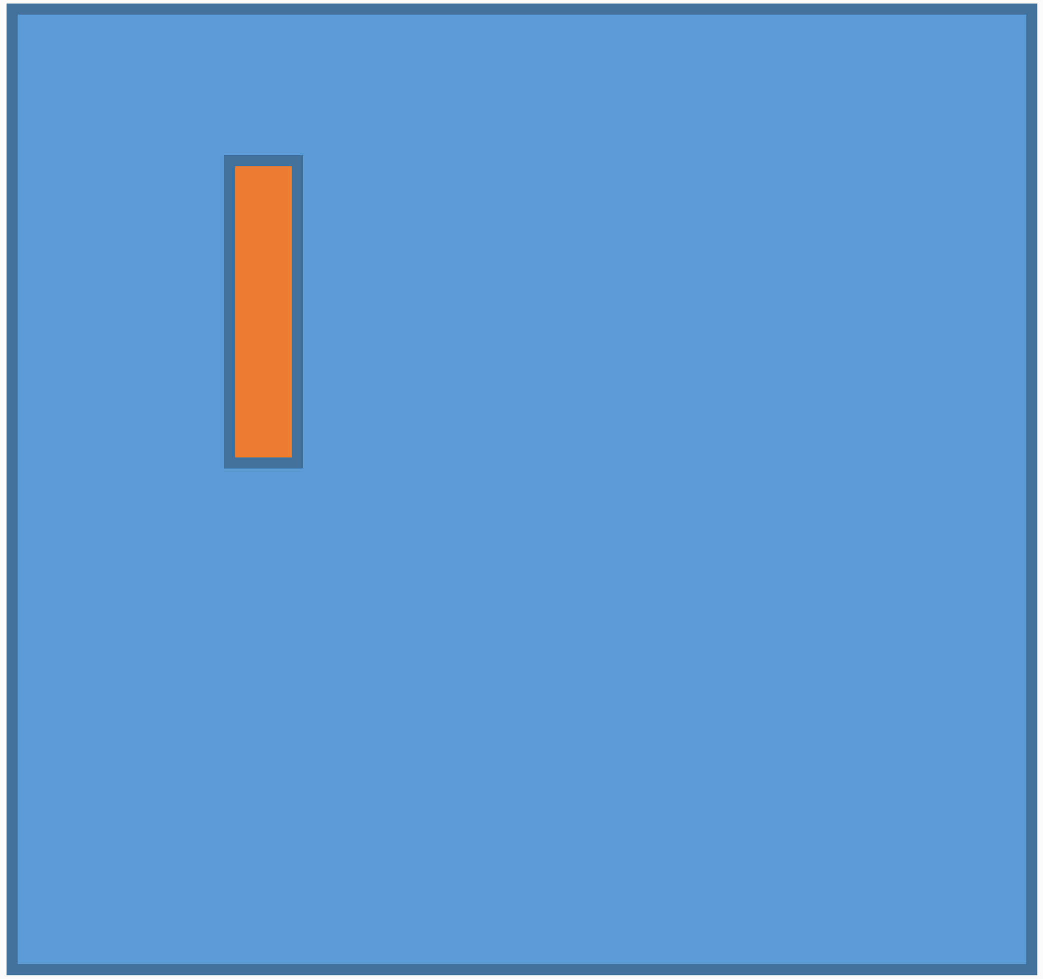
- A major problem is that the programming model does not correspond to the performance model
  - Programmer doesn’t see the cost of the constructs
  - It appears that all memory is “shared” and equally accessible
    - But the costs are affected by another processor’s actions
- You must be aware of communication via cache lines:
  - If a processor writes data and another reads it, it’s a communication, with associated costs, even if it’s a shared memory hardware
    - It takes time (compared with L1 cache, say)
    - It creates contention on the shared bus or communication network, causing additional delays

Example: Jacobi Relaxation

- Consider the problem of finding the temperature at every point in a room (or a square plate), where the temperature at the edges is 0°C and temperature of the heating element somewhere in the room is 100°C
- Problems like this, including those involving electric potential or even shape of soap bubbles on metal wireframes, can be solved using Laplace's equation or its generalization, the Poisson equation
- Numerically, there are several algorithms for solving it in a discretized grid
  - We will focus on Gauss-Jacobi Relaxation
  - This is an example of a large class of algorithms called iterative solvers
  - It is also an example of an important class of computation called stencil computations

The Jacobi Relaxation Algorithm

- The space is discretized into an N x N grid\
- The iterative step, that is applied repeatedly, updates the value (*temperature*) at each grid point as its average of its neighboring four points on itself
- The boundary condition – i.e., the fixed temperatures at the heating element and the edges – is enforced every step.
- The iterative computation continues until there is no significant change in temperature at any point



Jacobi with OpenMP:

- The inner loop nest, where most of the work is, is shown below

```
#pragma omp parallel for private(x,y)
for(x=1; x<MATSIZE-1; x++) {
    for (y=1; y<MATSIZE-1; y++) {
      newA[x][y] = (oldA[x][y] + oldA[x+1][y] + oldA[x-1][y] +
                   oldA[x][y+1] + oldA[x][y-1])/5;
    }
}
```

Jacobi with OpenMP: with outer iteration

- Different computation schemes
  - Which dimension to iterate over first, X or Y?
  - Which dimension to parallelize, X or Y?

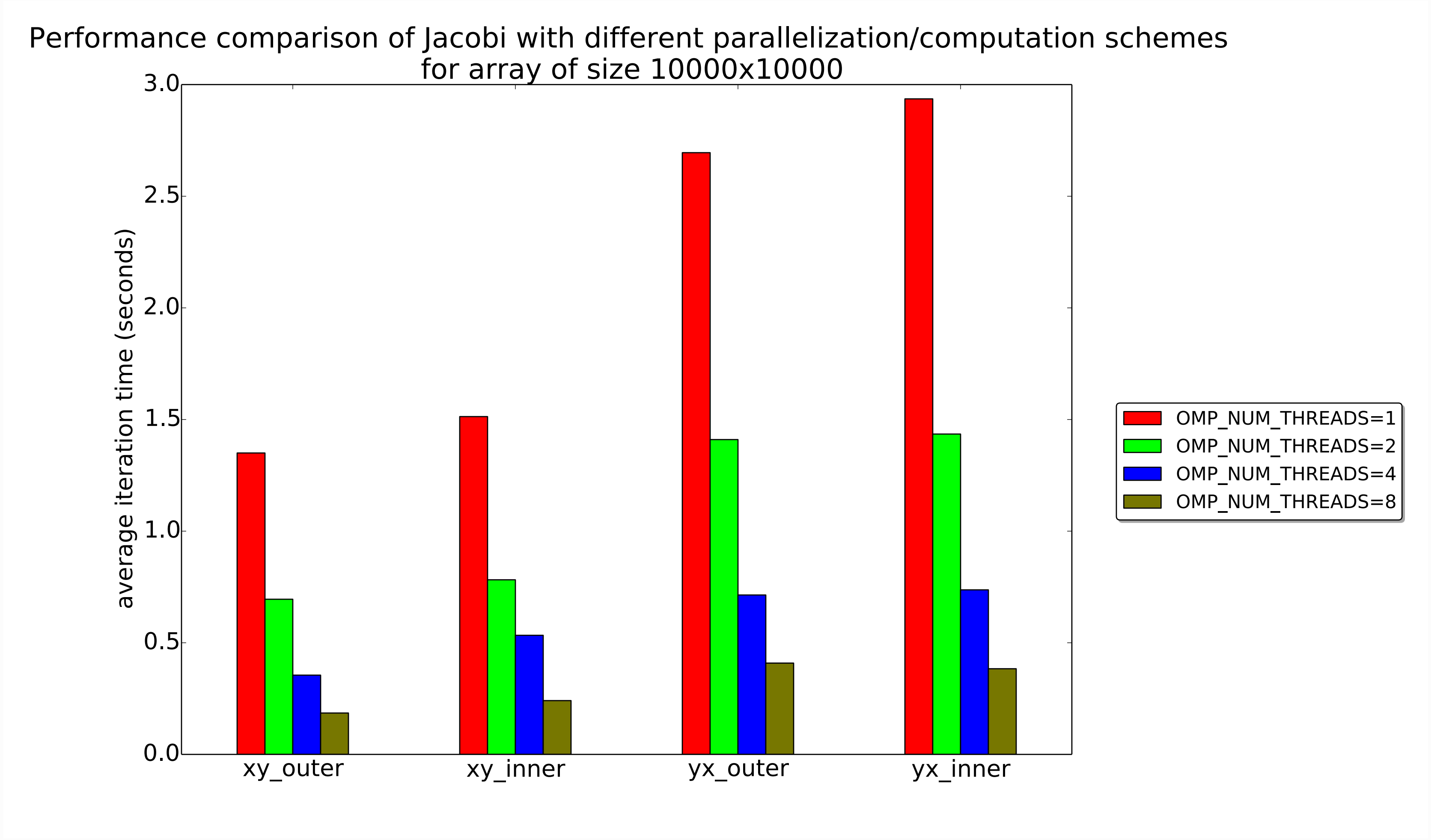
```
while (maxDelta > THRESHOLD) {
#pragma omp parallel for private(x,y), reduction(maxDelta)
  for(x=1; x<MATSIZE-1; x++) {
    for (y=1; y<MATSIZE-1; y++) {
      newA[x][y] = (oldA[x][y] + oldA[x+1][y] + oldA[x-1][y] +
                   oldA[x][y+1] + oldA[x][y-1])/5;
      float delta = abs(newA[x][y] - oldA[x][y]));
      if (delta > maxDelta) maxDelta = delta;
    }
 }
 swap newA and oldA pointers
}
```

Jacobi with OpenMP: focus on a step

- Different computation schemes
    - Which dimension to iterate over first, X or Y?
    - Which dimension to parallelize, X or Y?

```
#pragma omp parallel for private(x,y)
for(x=1; x<MATSIZE-1; x++) {
     for (y=1; y<MATSIZE-1; y++) {
       newA[x][y] = (oldA[x][y] + oldA[x+1][y] + oldA[x-1][y] +
                     oldA[x][y+1] + oldA[x][y-1])/5;
     }
}
```

Different computation schemes



Different way of expressing the same parallelization scheme (1)

- Take xy_inner as an example
  - overhead of creating omp threads in every inner loop
- Implicit → Explicit parallelism expression
  - Removes the above overhead

```
for(x=1; x<MATSIZE-1; x++) {
   #pragma omp parallel for private(y)
   for (y=1; y<MATSIZE-1; y++) {
     newA[x][y] = (A[x][y] + A[x+1][y] +
                   A[x-1][y] + A[x][y+1] +
                   A[x][y-1])/5;
   }
}
```

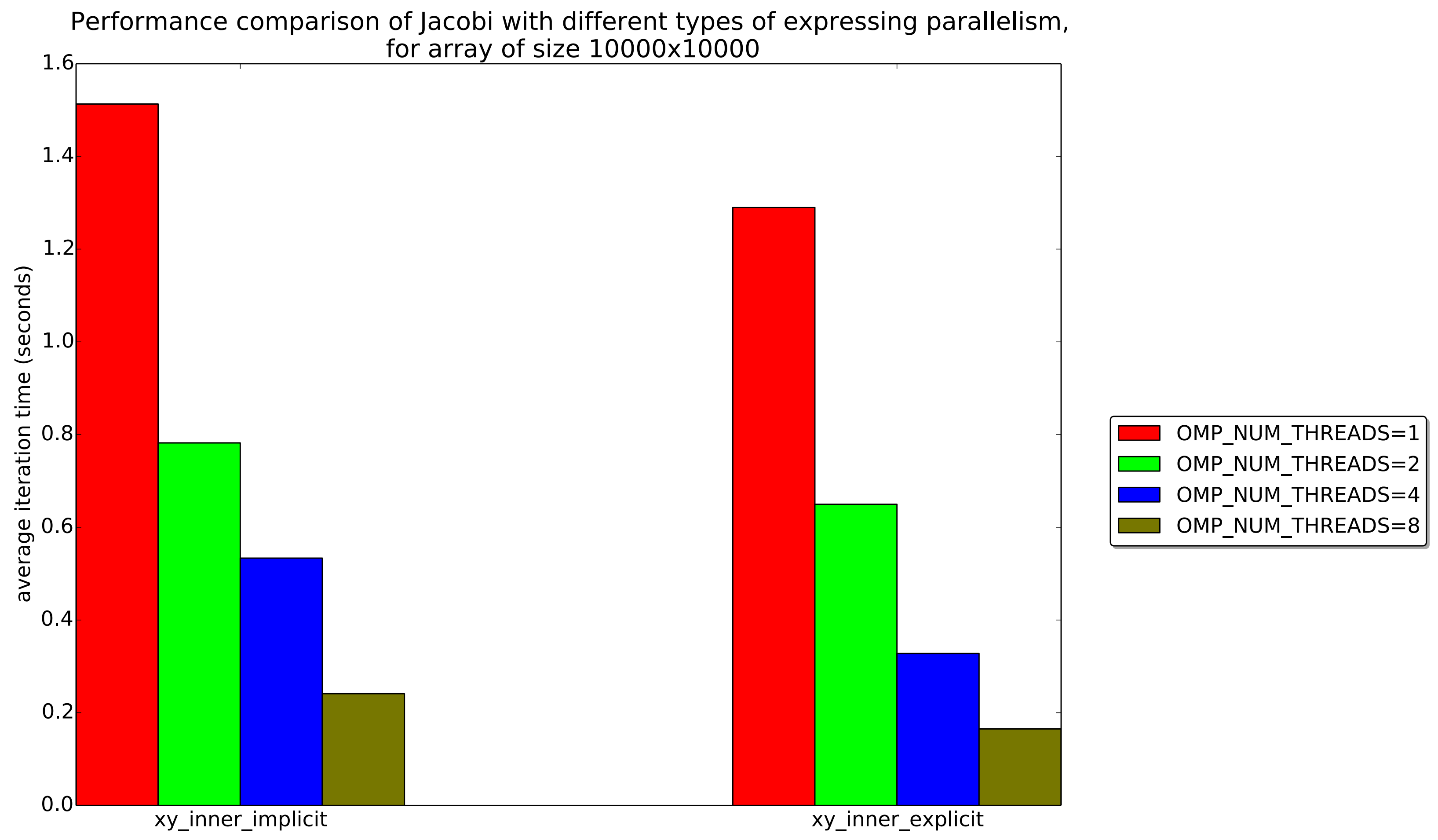
Implicit

→

```
#pragma omp parallel private(x,y)
{
   int chunksize = (MATSIZE-2)/omp_get_num_threads();
   int mystart = 1 + omp_get_threads_num()*chunksize;
   for(x=1; x<MATSIZE-1; x++) {
      for(y=mystart; y<mystart+chunksize; y++) {
         newA[x][y] = (A[x][y] + A[x+1][y] +
                       A[x-1][y] + A[x][y+1] +
                       A[x][y-1])/5;
      }
   }
}
```

Explicit

Different way of expressing the same parallelization scheme (2)

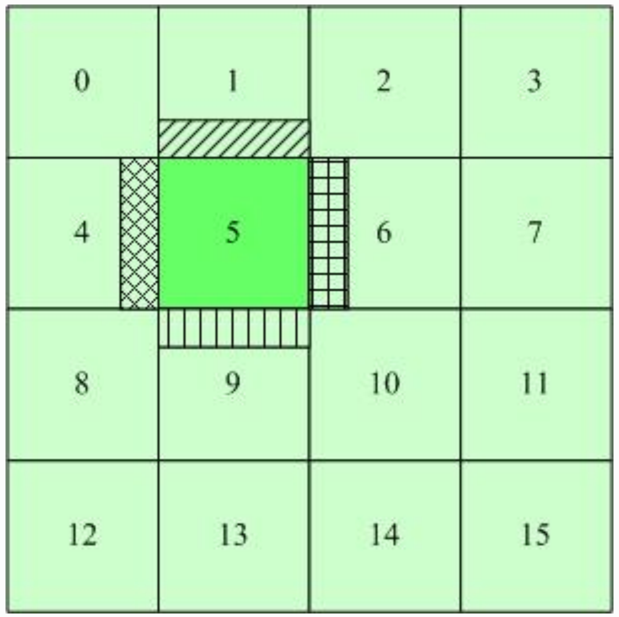


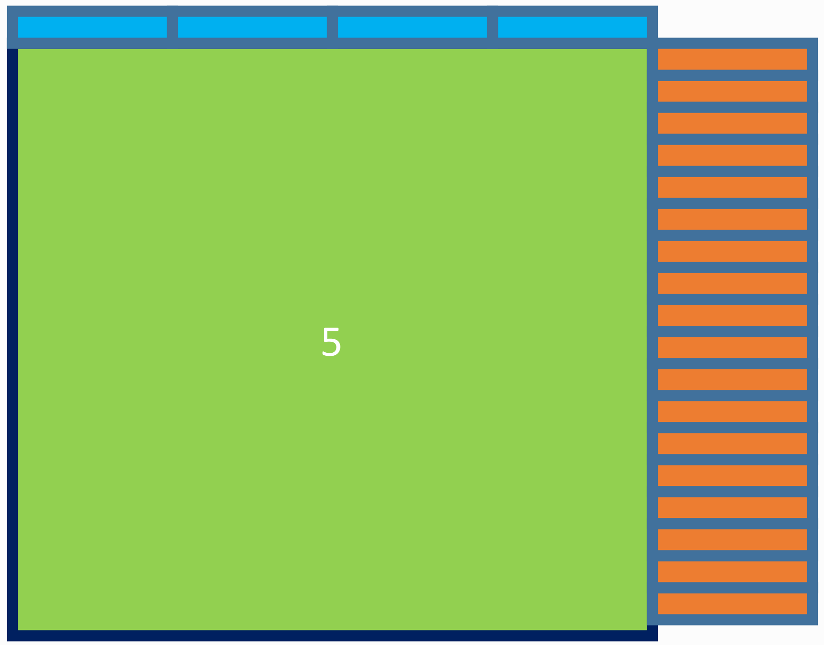
Parallelization for Absolute Performance

- Implicit barrier for each parallel construct in OpenMP
  - Each iteration of the outermost loop is separated by an implicit OpenMP barrier
- Is it possible that one thread starts the next iteration without waiting for all other threads to finish the current iteration?
  - Removing the barrier could lead to the overlap of computation from different iterations, thus saving time!

Is Square Decomposition Good?

Instead of parallelizing the first or second loop





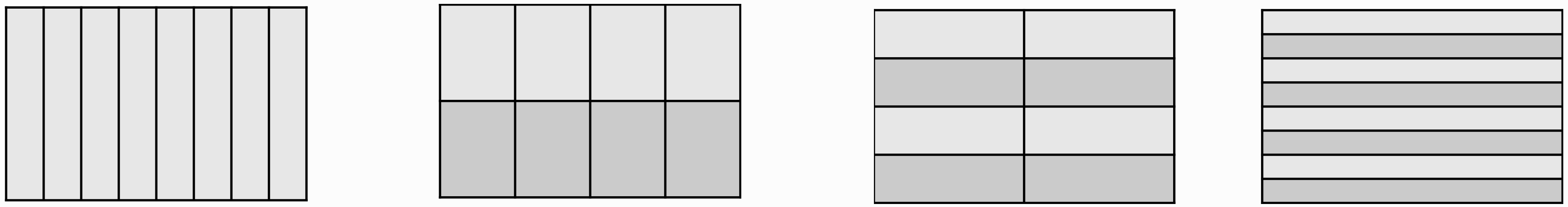
Not quite, because the situation is asymmetric across dimensions

It should be a rectangle with longer dimension along rows ...

Parallelization for Scalability

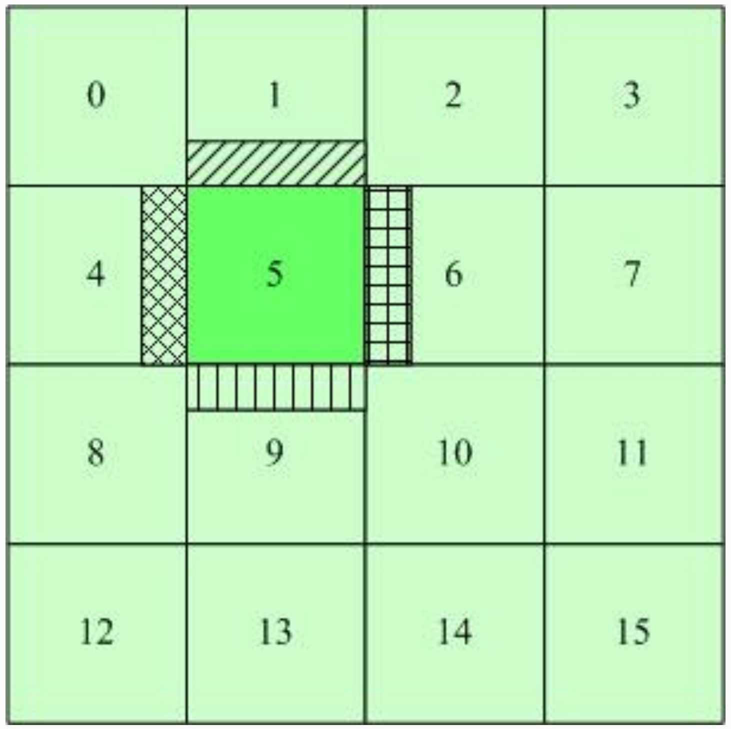
```
#pragma omp parallel for private(x,y,xx,yy,i)
for(i=0; i<numBlkX*numBlkY; i++) {
    xx = 1+(i/numBlkY)*BLKX;     /*BLKX is #elems in X-dim of tile*/
    yy = 1+(i%numBlkY)*BLKY;    /*BLKY is #elems in Y-dim of tile*/
    for(x=xx; x<xx+BLKX; x++) {
        for(y=yy; y<yy+BLKY; y++) {
            newA[x][y] = (oldA[x][y] + oldA[x+1][y] + oldA[x-1][y] +
                          oldA[x][y+1] + oldA[x][y-1])/5;

        }
    }
}
```

Alternatively, use “omp parallel” to explicitly partition work according to the following picture, assuming 8 cores



Parallelization for Absolute Performance

- Considering block decomposition of the matrix
- For simplicity, each thread holds one block



- Observation:
  - Thread 5 can start the next iteration when its neighbor threads 1,4,6,9 finish their updates for the shaded parts, respectively
  - Thread 5 doesn't need to wait for its non-neighbor threads (such as 0,3,7,13, etc.) to finish the current iteration to start the next iteration
  - How to do this? Use flags to signal readiness, e.g., using flush primitive

Some Lessons for Good Performance

- Sequential cache performance issues are still important
    - E.g., In Jacobi relaxation, xy order was better than yx
- All things being equal, parallelizing outer loop is better than inner loop
- You can regain efficiency using parallelization of inner loop by using "**`omp parallel`**"
    - Avoids thread creation and synchronization overhead
- Communication analyses, to see how much data created by one thread is read by another, is useful
    - And can be optimized by techniques such as block/tile decomposition
- Eliminating global barriers is important

Gauss-Seidel

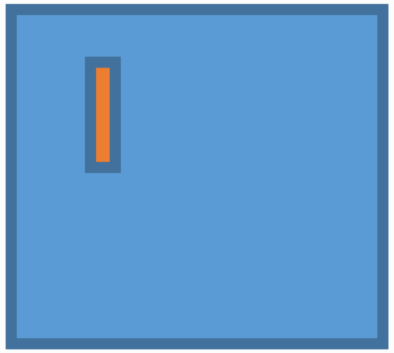
Dealing with Complicated Dependencies

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Gauss-Seidel Relaxation

Sequential pseudocode:

```
while (maxError > Threshold) {
  Re-apply Boundary conditions
  maxError = 0;
  for i = 0 to N-1 {
    for j = 0 to N-1 {
      old = A[i, j]
      A[i, j] = 0.2 * (A[i,j] + A[i,j-1] +A[i,j+1]
                     + A[i+1,j] + A[i-1,j]) ;
      if (|A[i,j]-old| > maxError)
        maxError = |A[i,j]-old|
    }
  }
}
```



For the same problem we solved using Jacobi Relaxation

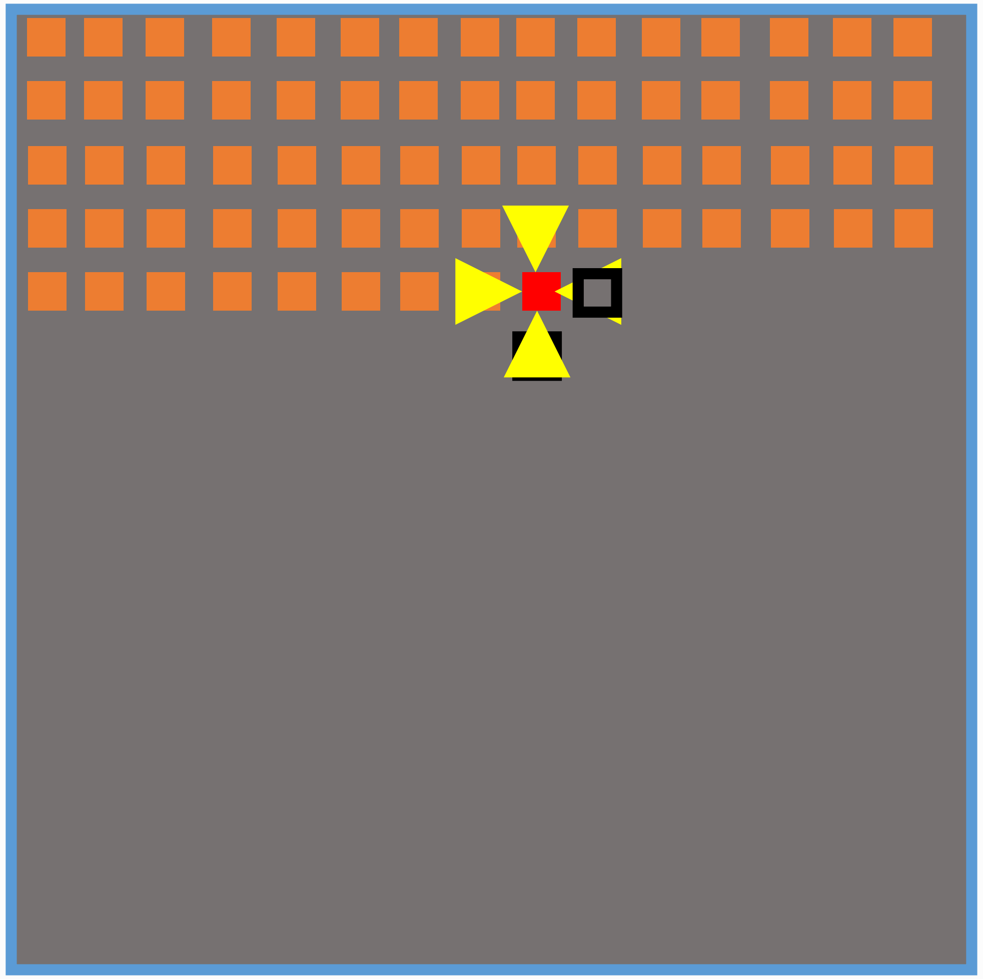
No old-new arrays ...

Sequentially, how well does this work?

It works much better!

- Intuitively, the effect of boundary conditions spreads fast to other areas, compared with Gauss-Jacobi

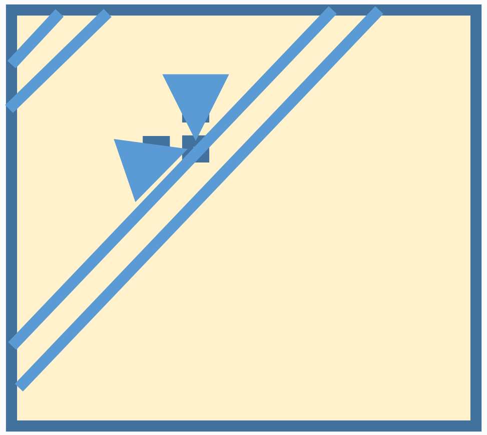
How to parallelize this?



```
A[i, j] = 0.2 * (A[i,j] + A[i,j-1] + A[i,j+1]
                          + A[i+1,j] + A[i-1,j]) ;
```

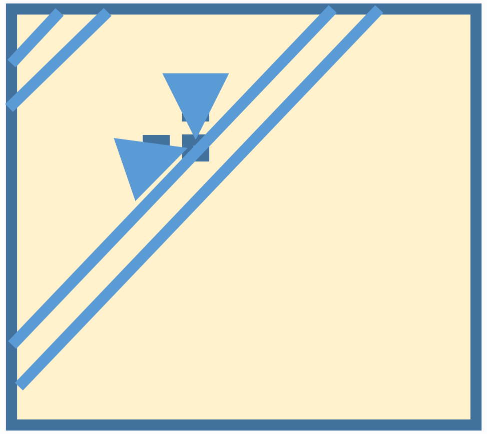
How Do We Parallelize Gauss-Seidel?

- Visualize the flow of values
- Not the control flow:
  - That goes row-by-row
- Flow of dependences: which values depend on which values?
- Does that give us a clue on how to parallelize?



How Do We Parallelize Gauss-Seidel?

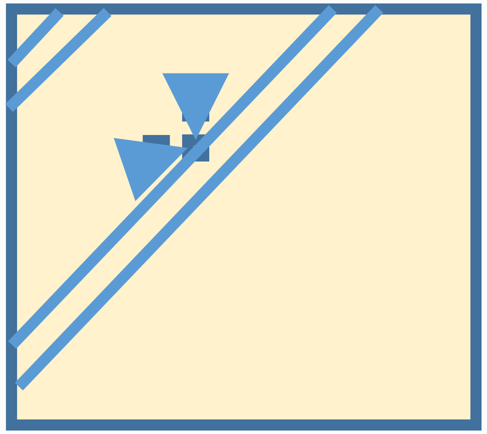
- Visualize the flow of values
- Not the control flow:
  - That goes row-by-row
- Flow of dependences: which values depend on which values?
- Does that give us a clue on how to parallelize?



```
for diagonal = 0 to 2*N-2 {
 parallel loop over values in the diagonal
  { i= .. ; j = ..;
    old = A[i,j];
    A[i, j] = … ;
    if (|A[i,j]-old| > maxError)
      maxError = |A[i,j]-old|
    }
  }
```

Gauss-Seidel: parallelize each diagonal

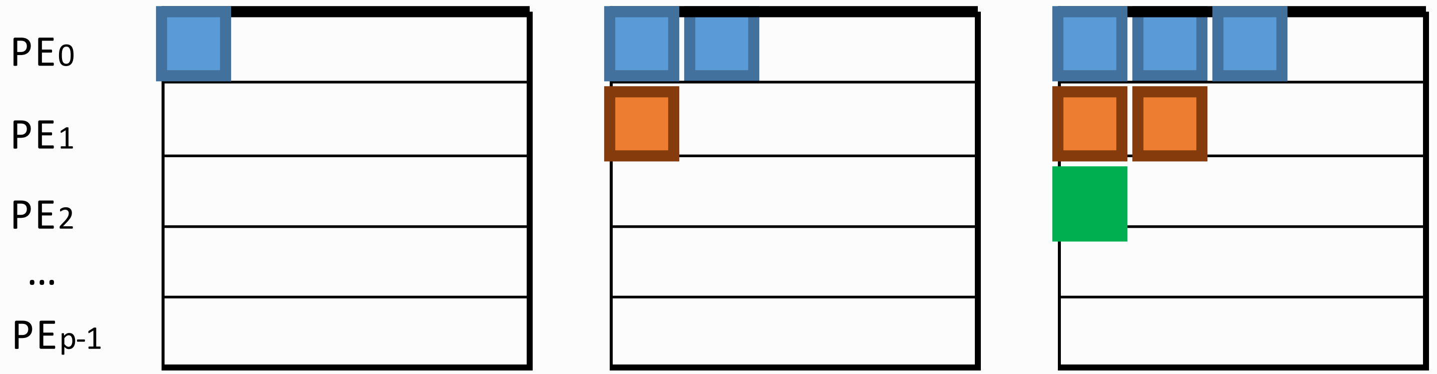
- Performance is not so good. Why?
- Each thread is doing a different (shifting) section of rows.
  - Spatial locality and prefetch efficiency is affected
- Too fine grained a loop? There are 2N parallel loops
- Other reasons? Implement and analyze with PAPI or perf tools



```
for diagonal = 0 to 2*N-2 {
 parallel loop over values in the diagonal
  { i= .. ; j = ..;
    old = A[i,j];
    A[i, j] = … ;
    if (|A[i,j]-old| > maxError)
      maxError = |A[i,j]-old|
    }
  }
```

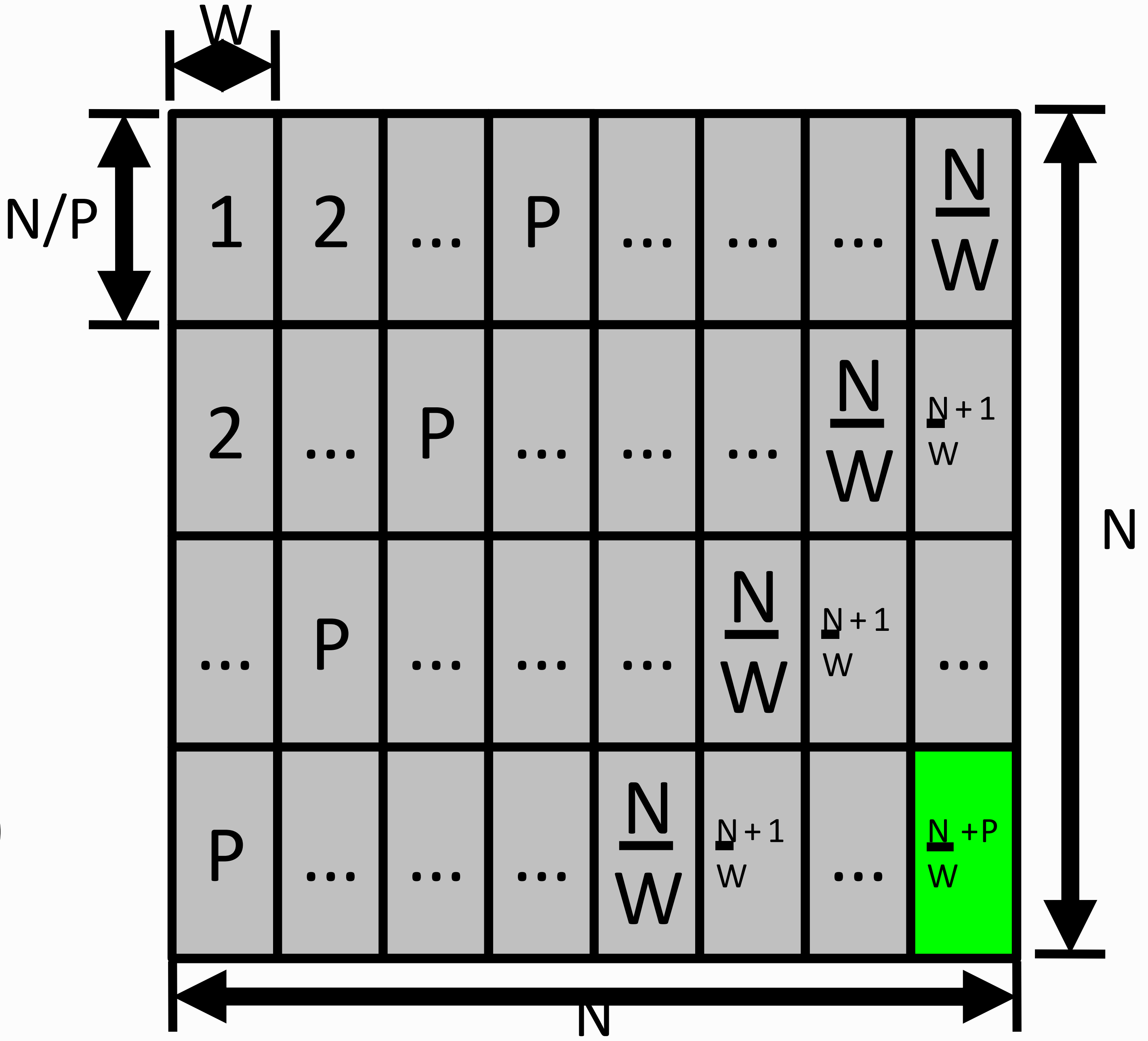
Parallelizing Gauss-Seidel

- Some ideas
  - Row decomposition, with pipelining



  - Square over-decomposition
    - Assign many squares to a processor (essentially same?)

Row decomposition with pipelining

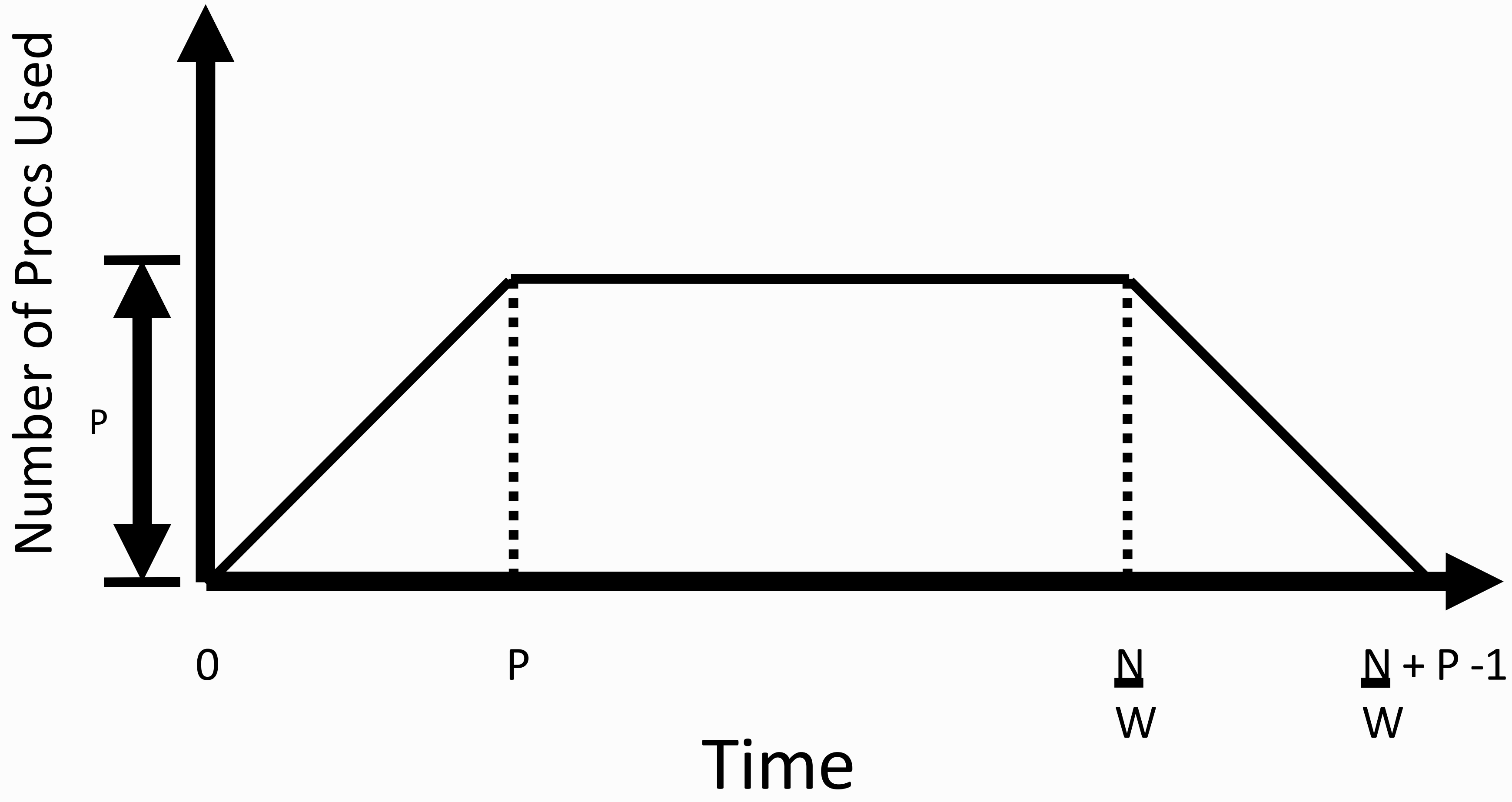


# Columns = N/W
# Rows = P

# Of Phases

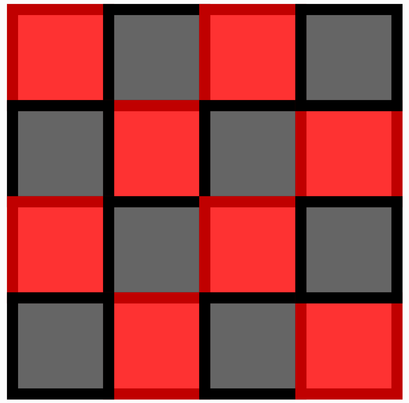
$N/W + (P-1)$

Row decomposition, with pipelining



Red-Black Squares Method

- Red squares calculate values based on the black squares
  - Then black squares use values from red squares
  - Now red ones can be done in parallel, and then black ones can be done in parallel
- A “square” may be just a single point
  - Or it can be a kxk tile of values
    - Each tile locally can do Gauss-Seidel computation
    - Faster convergence of Gauss-Seidel



False Sharing

Cache Performance Issues in Parallel Programming

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Consider the Following Code

- The code finds the index at which the largest value resides
- This will have pretty bad performance because threads will be hitting the critical section continuously, leading to serialization

```
curMax = MINUS_INFINITY; maxIndex = -1;
#pragma omp parallel for
for(i=0; i<n; i++){
#pragma omp critical
  {
    if (a[i] > curMax)
      { curMax = a[i]; maxIndex = i}
  }
}
```

Maximum Index: Improved

- The problem is: in every iteration, we enter the critical section
  - What if we enter the critical section only after the test?
  - This might be incorrect because another thread might have changed the cur_ max in the meanwhile
- We repeat the test but it is correct now
- Further, we hit the critical section less often, reducing contention and serialization

```
curMax = MINUS_INFINITY; maxIndex = -1;
#pragma omp parallel for
for(i=0; i<n; i++){
  if (a[i] > curMax)
  #pragma omp critical
  {
    if (a[i] > curMax)
      { curMax = a[i]; maxIndex = i}
  }
}
```

Maximum Index: Better Strategy

- The previous code still has a potential for significant contention for the critical section
- We can avoid this by privatizing maxIndex, having each thread keep track of it in the parallel region, and selecting the maximum from among each thread’s maximum at the end
    - We will explicitly privatize maxIndex, by using an array maxIndices[] indexed by thread ID

Maximum Index Using Private Variables

```
int* maxIndices = new int[p];
//p is number of threads
#pragma omp parallel
{
 int id = omp_get_thread_num();
 maxIndices[id] = 0;
#pragma omp for
  for(i=0; i<n; i++)
     if (a[i] > a[ maxIndices[id] ])
       maxIndices[id] = i;
}
maxIndex = maxIndices[0];
  for(i=1; i<p; i++)
    if (a[maxIndices[i]] > a[maxIndex])
      maxIndex = a[maxIndices[i];
// now maxIndex is the index of the
largest value in the array
curMax = a[maxIndex];
```

We expect this code to have good performance because there is no critical section in the main loop

And the second loop executed by the master is short

But we find the performance is not very good

Why?

The Problem: Cache Traffic Increase

- The problem with the code is that different threads are writing to the same cache line
  - When they write to **`maxIndices[i]`**
  - The cache line is continuously shuttled between caches of different cores
  - This is sometimes called "thrashing"
- This is inspite of the fact that each thread writes only to its own location – **`maxIndices[id]`**
  - I.e., there is no sharing of data
- This is called false sharing
- A common solution to this problem is "padding," so that each thread writes to its own cache line

```
typedef struct{
int index;
char padding[28]
}
PaddedIndex;
```

`maxIndices[]` is now an array of these structures
`maxIndices[id].index` should be used instead of
`maxIndices[id]` in the code of the previous slide

This assumes a 32-byte cache line

Recap: False Sharing

- False sharing happens when:
  - Two or more threads access the same cache line for writes
  - Over a common time interval
  - Writing to distinct variables
    - I.e., no two threads are writing to the same variable
    - So, there is no real sharing
  - But because write access involves bringing the whole cache line to my private (say L1) cache, there is continuous traffic, leading to performance loss
  - This is “unnecessary” in the sense that if cache line size was one word, it would not happen
- One technique for eliminating false sharing is padding
  - So, data accessed by distinct threads is in different cache lines