CS420 – Lectures 10 and 11

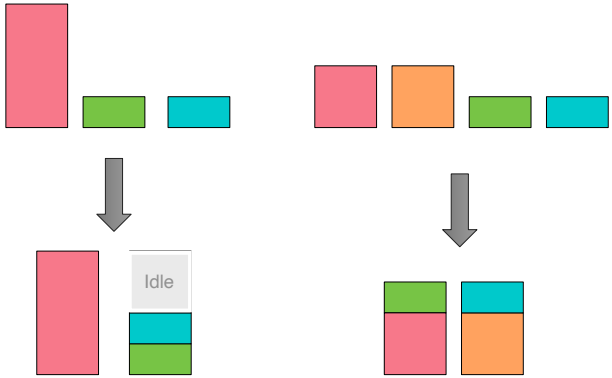
Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Parallel Loops

Work sharing

- The `parallel` construct assigns an *implicit task* (the execution of a structured block of code) to each thread.
  - Good: Programmer is in full control of what each thread executes
  - Bad: Programmer needs to fully control what each thread executes
- *Load balancing:* Ensure that each thread has equal amount of work (assuming they all run at same speed)



Work sharing

Assume there is a bag of independent tasks to execute in parallel

- Allocate them to threads statically (as for sum of square example)
  - Good if we know how long each task will run
- Allocate them dynamically, at run time
  - Let the system do it (use *work sharing* constructs)

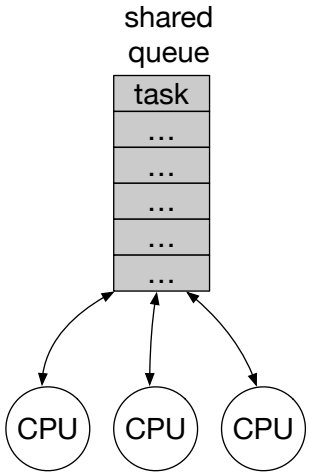
```
...
#pragma omp parallel for \
                reduction(+:sum)
  for(int i=0; i<N; i++)
    sum+=i*i;
...
```

- The system will allocate iterates of the loop to running threads using some scheduling policy
  - Loops need be of simple form

How is work sharing done?

Simplemost: Shared work queue

- Parallel section start: iterates are (virtually) queued
- Threads pick work to execute from queue
- Parallel section ends when queue is empty and all threads are done (each tried to get work from empty queue)



Work sharing tradeoffs

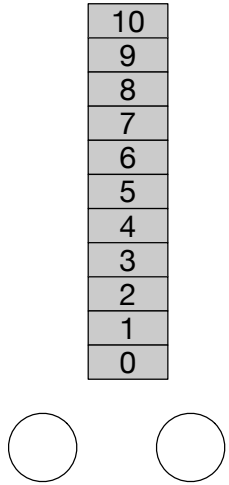
- Need tasks large enough in order to amortize the overhead of scheduling a task
  - Rule of thumb: 1000's of instructions; one iteration in our example is much too small
- Need tasks small enough so that load balancing works well
  - Rule of thumb: If execution time of tasks is not fixed, then number of tasks should be a small multiple of number of threads: *over-decomposition*

Loop schedule

- Allocation of iterates to threads is controlled by a `schedule` clause

```
...
  #pragma omp parallel for \
      schedule(static, chunk_size) \
     reduction(+:sum)
    for(int i=0; i<N; i++)
      sum+=i*i;
 ...
```

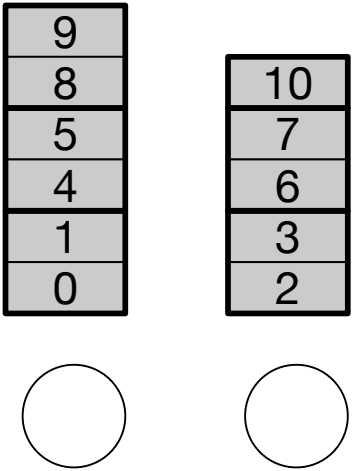
Assume $N = 11$, numthreads=2. Tasks (iterates) are ordered in queue



Static

```
...
  #pragma omp parallel for \
      schedule(static,2)
    for(i=0;i<N;i++)
        ...
```

- Iterates are allocated round robin, in chunks of 2, to the threads
- Best when iterates (and threads) are all the same – low scheduling overhead
- Same allocation at each execution (with same number of threads)

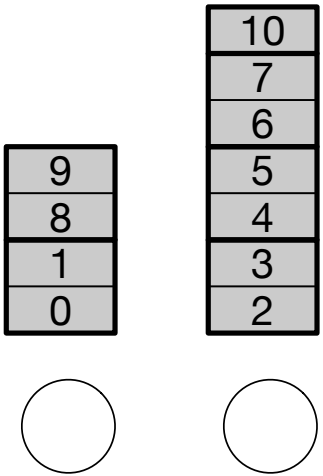


Dynamic

```
...
  #pragma omp parallel for \
      schedule(dynamic,2)
    for(i=0;i<N;i++)
        ...
```

- threads are dynamically picking iterates, in chunks of 2
- Best for load balancing, but higher scheduling overhead

One possible execution



Guided

```
...
  #pragma omp parallel for \
      schedule(guided,2)
     for(i=0;i<N;i++)
        ...
```

- threads are dynamically picking iterates, in chunks of at least two. Number of iterates picked is proportional to number of iterates left divided by numthreads
- Compromise between static and dynamic

Auto, runtime

```
...
  #pragma omp parallel for \
      schedule(auto)
    for(i=0;i<N;i++)
       ...
```

- I have no idea what's the right schedule; compiler and runtime will do what they think best

```
...
  #pragma omp parallel for \
      schedule(runtime)
    for(i=0;i<N;i++)
       ...
```

- I delay to runtime the choice of the schedule

What's the difference between auto and runtime?

- Various OpenMP *Internal Control Variables* (ICV) can be set at runtime. Example: number of active threads
  - Environment variable `OMP_NUM_THREADS` can be set before an OpenMP program starts executing; this will be the default number of threads for the program execution
  - `omp_set_num_threads()` can be called inside a program; it will set the number of threads for current team
  - `omp_get_num_threads()` can be called to query the current value of the ICV.
- Example: default schedule for parallel loops
  `OMP_SCHEDULE`, `omp_set_schedule(sched, chunk_size)` and `omp_get_schedule(sched, chunk_size)`
- `schedule(runtime)` will use the schedule defined by the current ICV value (same as a loop with no schedule clause)

```
N=10;
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    for (k=0; k < N; k++)
      A[i][j][k] = f(i, j, k);
      // assume f is expensive but
      // variable in execution time
```

- Assume that you have a machine with 30 cores
- All the loops are parallel, but none is adequately large enough to employ all the threads

```
N=10;
#pragma omp parallel for schedule(dynamic)
for (m = 0; m < N*N*N; m++) {
  i = m / (N*N);
  j = (m % (N*N)) / N;
  k = m%N;
  A[i][j][k] = f( i, j, k);
  // assume f is expensive but
  // variable in execution time
}
```

- Assume that you have a machine with 30 cores
- All the loops are parallel, but none is adequately large enough to employ all the threads

Nested Parallelism

```
N=10;
#pragma omp parallel for collapse(3) schedule(dynamic)
for (i=0; i< N; i++)
  for (j=0; j< N; j++)
    for (k=0; k< N; k++)
      A[i][j][k] = f(i, j, k);
      // assume f is expensive but
      // variable in execution time
```

- 3 nested loops are combined to make a single loop with 1000 iterations.
- i, j, k calculated automatically.
- Also, no need to declare j and k private. They are implicitly private

Nested Parallelism

```
int p;
omp_set_nested(1);
// omp_set_max_active_levels();
omp_set_dynamic(0);
#pragma omp parallel num_threads(8)
{
  #pragma omp single
  printf("outer total number of omp threads = %d\n", omp_get_num_threads());

  printf("thread number: %d\n", omp_get_thread_num());

  #pragma omp parallel num_threads(2)
  printf("inner parallel region thread number: %d\n", omp_get_thread_num());
}
```

What happens with nested parallelism?

- Might not be supported (get error) – controlled by ICV
  - `OMP_MAX_ACTIVE_LEVELS`, `omp_set_max_active_levels`, `omp_get_max_active_levels`
- Even if it is supported it is not obvious how many threads will execute a nested loop – could be one

Nested parallelism

Note:

```
#pragma omp parallel for
```

is equivalent to

```
#pragma omp parallel
#pragma omp for
```

first statement creates team; second statement does work sharing across team

Examples

Matrix Product

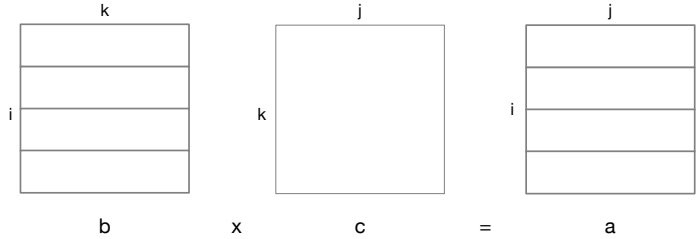
- Can allocate to each processor a tile to compute – provided the computation of distinct tiles are independent
- If tile in `k` dimension need to add a reduction.
- Always want to parallelize outermost loop (get large tasks)

Tile i

```
#include <omp.h>
#include <stdio.h>
#define N 500
double a[N][N],b[N][N],c[N][N];
int main(int argc, char *argv[]) {
  int i, j, k, n;
  double time;
  for (n=0; n<10; n++) {
    time = omp_get_wtime();
    omp_set_num_threads(4);
    #pragma omp parallel for
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        for (k=0; k<N; k++)
          a[i][j] += b[i][k] * c[k][j];
    printf("%d ", (int)(1000.0 *
              (omp_get_wtime()-time)));
  }
  printf("\n");
}
```

- `omp_get_wtime` returns time in seconds.
- Only outermost (`i`) loop is executed in parallel

Different tiling = different partitions



Tile `i`: Each thread computes product of horizontal tile of `b` with `c` that yields a horizontal tile of `a`.

Tile j

```
#pragma omp parallel for
for (j=0; j<N; i++)
  for (i=0; i<N; j++)
    for (k=0; k<N; k++)
      a[i][j] += b[i][k] * c[k][j];
```

Tile k

```
#pragma omp parallel for \
   reduction(+:a[:][:])
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] += b[i][k] *c [k][j];
```

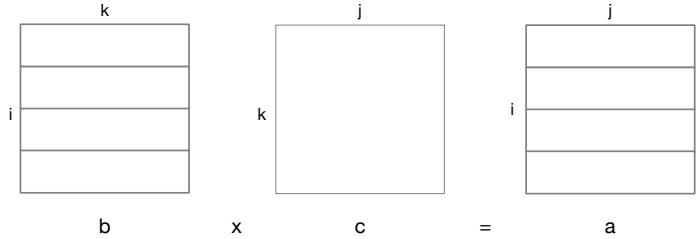
- The `reduction` clause takes an array section argument (will be discussed later)

Tile both i and j

```
#pragma omp parallel for collapse(2)
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      a[i][j] += b[i][k] * c[k][j];
```

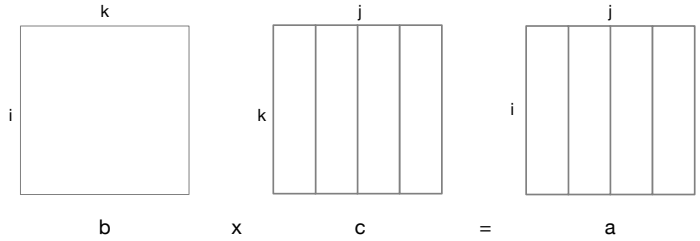
- `collapse(2)` indicates that two outermost loops should be taken as one loop (with N×N iterates) and executed in parallel

Different tiling = different partitions



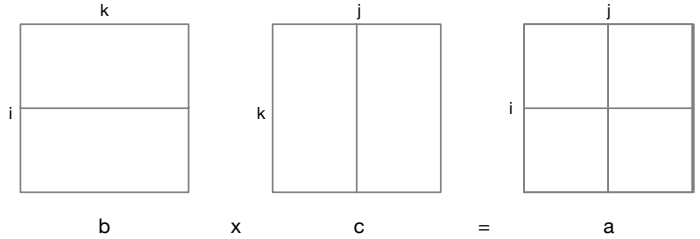
Tile `i`: Each thread computes product of horizontal tile of `b` with `c` that yields a horizontal tile of `a`.

Different tiling = different partitions



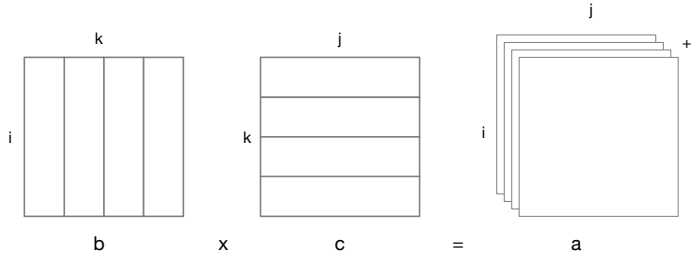
Tile `j`: Each thread computes product of `b` with vertical tile of `c` that yields a vertical tile of `a`.

Different tiling = different partitions



Tile `i,j`: Each thread computes product of horizontal tile of `b` with a vertical slice of `c` that yields a 2D tile of `a`.

Different tiling = different partitions



Tile `k`: Each thread computes product of vertical tile of `b` with a horizontal slice of `c` that yields an N×N matrix; the resulting matrices need to be added.

Running time

Code	Time (msec)
Tile `i`	1277±57
Tile `j`	1623 ±43
Tile `i,j`	992±14

- Tiling choices impact locality

Sequential Jacobi

```
   ...

do {
 err=0; l=1-k;
 for (i=1;i<M-1; i++)
  for(j=1;j<N-1;j++) {
   a[1-k][i][j]=0.25*(a[k][i-1][j]+a[k][i+1][j]+a[k][i][j-1]
       +a[k][i][j+1]);
   err = fmax(err,fabs(a[1][i][j]-a[0][i][j]));
  }
 } while(err>maxerr);
...
```

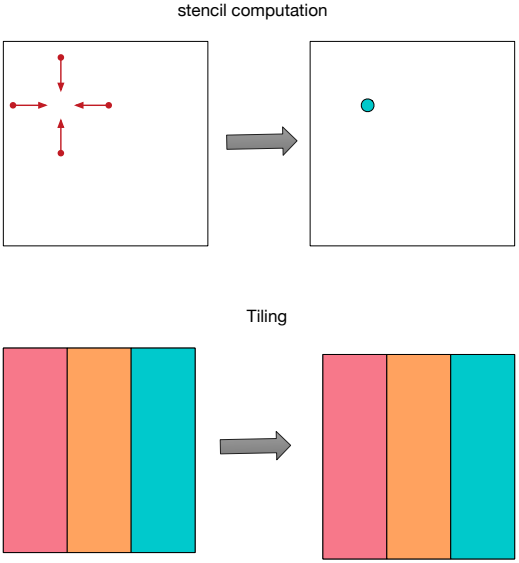
Parallel Jacobi

```
...
do {
 err=0; k=1-k;
#pragma omp parallel for collapse(2) reduction(max:err)
 for (i=1;i<M-1; i++)
  for(j=1;j<N-1;j++) {
   a[1-k][i][j]=0.25*(a[k][i-1][j]+a[k][i+1][j]+a[k][i][j-1]
        +a[k][i][j+1]);
   err = fmax(err,fabs(a[1][i][j]-a[0][i][j]));
  }
 } while(err>maxerr);
...
```

`collapse(2)`: the two outer loops are handled as one parallel loop with MN iterations

Possibly better

Tile the nested loops and allocate to threads full tiles



```
 ...
do {
 err=0; k=1-k;
 #pragma omp parallel for reduction(max:err)
 for(jj=1;jj<N-1;jj+=T)
  for (i=1;i<M-1; i++)
   for(j=jj;j<jj+T;j++) {
    a[1-k][i][j]=0.25*(a[k][i-1][j]+a[k][i+1][j]
         +a[k][i][j-1]
         +a[k][i][j+1]);
   err = fmax(err,fabs(a[1][i][j]-a[0][i][j]));
  }
 } while(err>maxerr);
 ...
```

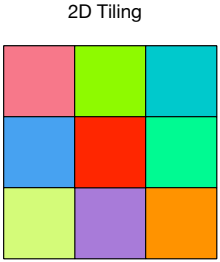
Only outer loop is executed in parallel

Tiling provides the same improvements in cache hit ratio as for sequential code

Assuming tiles are cache line aligned

What if number of tiles is too small?

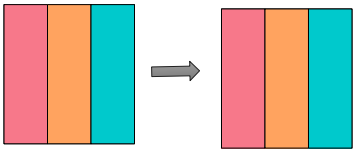
Solution 1: Use 2D tiles




```
\* N=n*T1+2, M=m*T2+2 *\
...
do {
  err=0;k=1-k;
 #pragma omp parallel for collapse(2) reduction(max:er
for(ii=1; ii<N-1;ii+=T1)
 for(jj=1;jj<M-1;jj+=T2)
  for (i=ii;i<ii+T2; i++)
   for(j=jj;j<jj+T2;j++) {
   a[1-k][i][j]=0.25*(a[k][i-1][j]+a[k][i+1][j]+a[k][i][j-1]
       +a[k][i][j+1]);
   err = fmax(err,fabs(a[1][i][j]-a[0][i][j]));
  }
} while(err>maxerr);
...
```

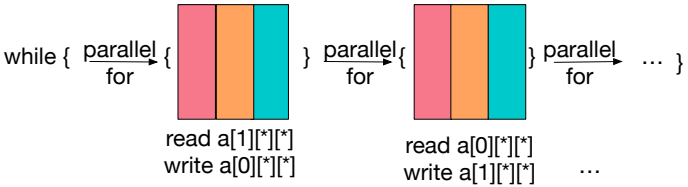
Jacobi

```
...
do {
  err = 0;
  k = 1-k;
  #pragma omp parallel for reduction(max:err)
  for (jj=1; jj<N-1; jj += T)
    for (i=1; i<M-1; i++)
      for (j=jj; j<jj+T; j++) {
        a[1-k][i][j] = 0.25 * (a[k][i-1][j] + a[k][i+1][j] +
                               a[k][i][j-1] + a[k][i][j+1]);
        err = fmax(err, fabs(a[1][i][j]-a[0][i][j]));
      }
} while (err > maxerr);
...
```



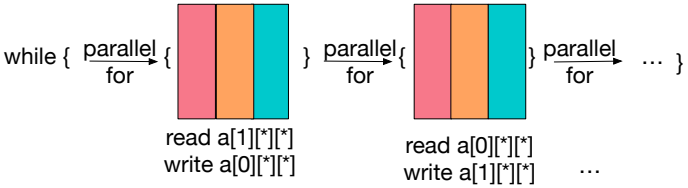
- Do we have races?

- Do we have races?
- No
  - `err` is a reduction variable
  - During each iteration of the while loop we read one copy of `a` and write another copy – no conflicts
  - No thread starts next iteration of the while loop before all threads completed the previous iteration – there is an implicit barrier at the end of the parallel section

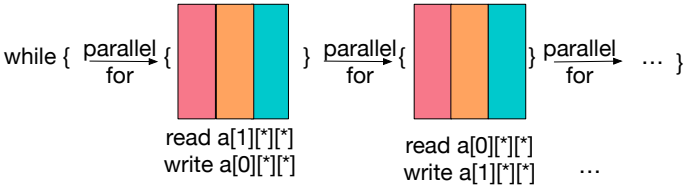


- Do we have communication between threads?

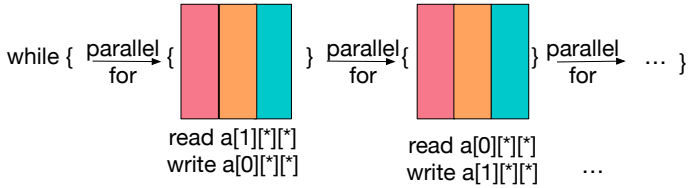
- Do we have communication between threads?
- Yes - “orange” thread needs a column written by each of “purple” and “blue” threads at previous “while” iteration, assuming threads pick same chunk at successive iterations.



- Do we have communication between threads?
- Yes - "orange" thread needs a column written by each of "purple" and "blue" threads at previous "while" iteration, assuming threads pick same chunk at successive iterations.
- Are we sure that a thread picks same slice at successive iteration?



- Do we have communication between threads?
- Yes - "orange" thread needs a column written by each of "purple" and "blue" threads at previous "while" iteration, assuming threads pick same chunk at successive iterations.
- Are we sure that a thread picks same slice at successive iteration?
- Not in general: allocation may change from parallel loop to parallel loop



Allocation does not change from one `parallel for` to the next if

- Number of threads is fixed
- Schedule is static

```
\* N=n*T+2 *\
...
omp_set_dynamic(0);
do {
  err = 0;
  k = 1-k;
  #pragma omp parallel for schedule(static) reduction(max:err)
  for (jj=1; jj<N-1; jj += T)
    for (i=1; i<M-1; i++)
      for (j=jj; j<jj+T; j++) {
        a[1-k][i][j] = 0.25 * (a[k][i-1][j] + a[k][i+1][j] +
                               a[k][i][j-1] + a[k][i][j+1]);
        err = fmax(err, fabs(a[1][i][j]-a[0][i][j]));
      }
} while (err > maxerr);
...
```

Jacobi "static" style

Can we avoid the overhead of repeatedly forking and joining control?

```
#pragma omp parallel
{
  n = omp_get_num_threads();  myid = omp_get_thread_num();
  myfirst = myid*N/n;         nextfirst = (myid+1)*N/n;
  do {
    #pragma omp single
    err = 0;
    myerr = 0; k = 1-k;
    for (i=1; i<M-1; i++)
      for (j=myfirst; j<nextfirst; j++) {
        a[1-k][i][j] = 0.25 * (a[k][i-1][j] + a[k][i+1][j] +
                               a[k][i][j-1] + a[k][i][j+1]);
        myerr = fmax(myerr, fabs(a[1][i][j]-a[0][i][j]));
      }
    #pragma omp critical
    err = fmax(err, myerr);
    #pragma omp barrier
  } while (err > maxerr);
}
```

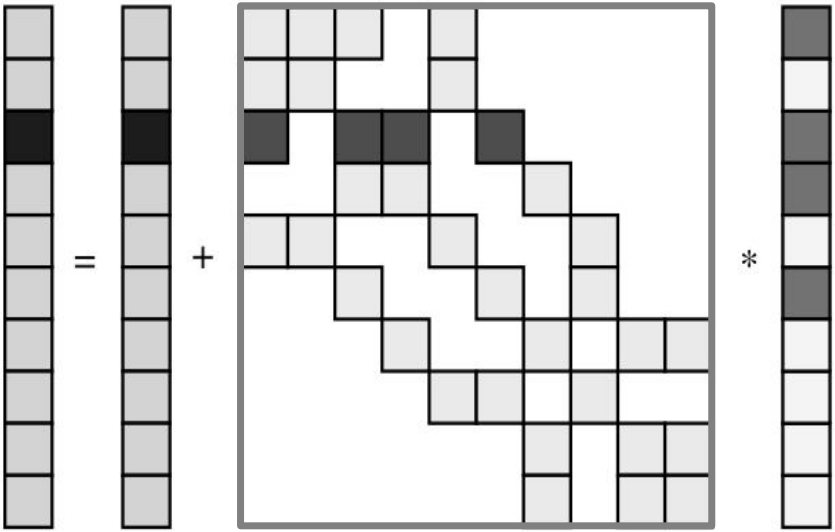
- Code has become more verbose.
- No load balancing by OpenMP runtime – user has to do it, if needed
- Cannot use `reduction`
- Cannot use atomic for max(in C/C++)
- May be better if starting/ending a parallel section is expensive

Have your cake and eat it too

```
#pragma omp parallel
do {
  #pragma omp single
  {
    err = 0;
    k = 1-k;
  }
  #pragma omp for collapse(2) reduction(max:err)
  for (i=1; i<M-1; i++)
    for (j=1; j<N-1; j++) {
      a[1-k][i][j] = 0.25 * (a[k][i-1][j] + a[k][i+1][j] +
                             a[k][i][j-1] + a[k][i][j+1]);
      err = fmax(err, fabs(a[1][i][j]-a[0][i][j]));
    }
  /* implicit barrier at end of omp for */
} while (err > maxerr);
```

Sparse data structures

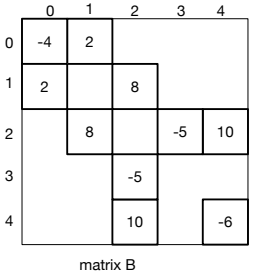
SparseMV: $a = b + Cd$ where C is a *sparse matrix*: most entries are zero.

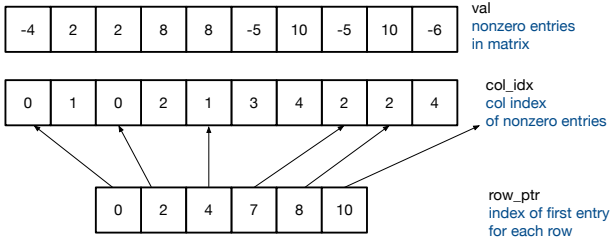


CRS

- How does one store the matrix so that only non-zeros are stored?
- How does one avoid the multiplications by zero?

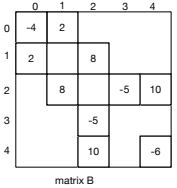
CRS: *Compressed Row Storage*

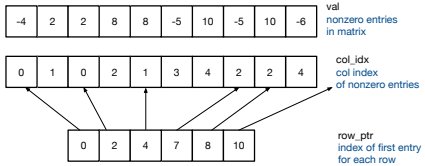




SparseMV

```
for(i=0;i<N;i++)
 a[i]=b[i];
  for(j=row_ptr[i];
      j<row_ptr[i+1];j++)
    a[i]+=val[j]*d[col_idx[j]]
```

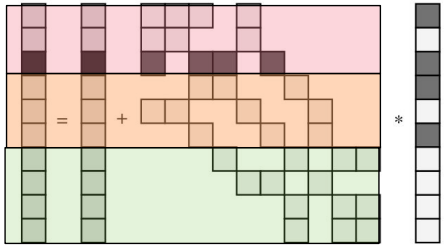




Parallel SparseMV

If rows have roughly the same number of non-zeros, then get good load balancing by statically tiling rows

```
#pragma omp parallel for \
     schedule(static,T)
  for(i=0;i<N;i++) {
   a[i]=b[i];
    for(j=row_ptr[i];
      j<row_ptr[i+1];j++)
       a[i]+=val[j]*d[col_idx[j]];
 }
```



T chosen so that tasks are large enough

Parallel SparseMV

If rows have very different number of non-zeros, need to use dynamic load balancing.

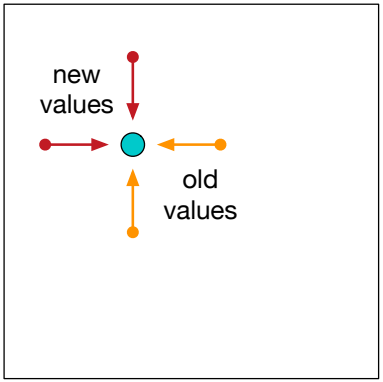
```
#pragma omp parallel for schedule(dynamic,T)
  {
  for(i=0;i<N;i++)
   a[i]=b[i];
    for(j=row_ptr[i];
      j<row_ptr[i+1]-1;j++)
       a[i]+=valj]*d[col_idx[j]];
  }
```

T chosen so that tasks are large enough, but number of tasks still large wrt number of threads

Gauss-Seidel

Like Jacobi, except done in place (one array)

$$a_{i,j}^{(k+1)} = 0.25(a_{i-1,j}^{(k+1)} + a_{i,j-1}^{(k+1)} + a_{i+1,j}^{(k)} + a_{i,j+1}^{(k)})$$



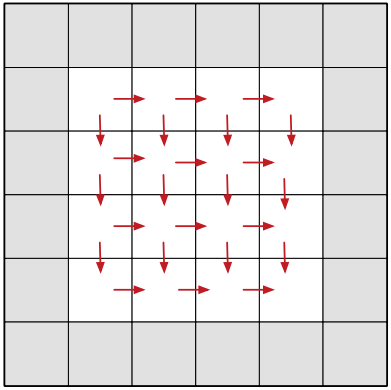
Sequential code

```
 ...
for(i=1;i<M-1;i++)
 for(j=1;j<N-1;j++)
  a[i][j]=0.25*(a[i-1][j]+a[i][j-1]
              +a[i+1][j]+a[i][j+1]);
...
```

How do we parallelize?
How can we reorder the nested loop so as to have many independent operations?

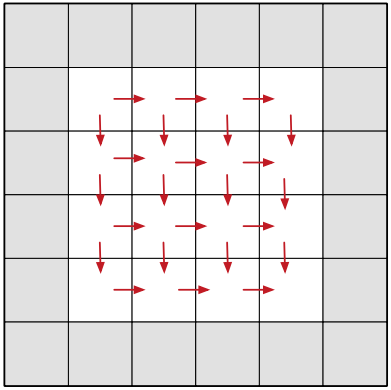
Wavefront

Loop carried dependencies

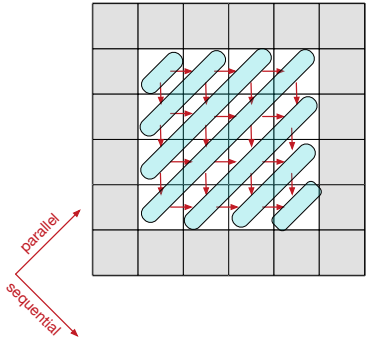


Wavefront

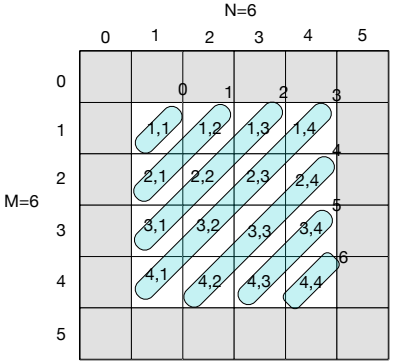
Loop carried dependencies



Wavefront



code




```
...
/* number of diagonals is M+N-5 */
for(d=0;d<M+N-5;d++) {
/* first & last diagonal row */
 ifirst=(d<M-1)?d+1:M-2;
 ilast=(d<N-1)?d-M+3:1;
 #pragma omp parallel for
  for(i=ifirst; i<=ilast;i++){
    j=d+2-i;
    a[i][j]=0.25*(a[i-1][j]
     +a[i+1][j]
     +a[i][j-1]+a[i][j+1])
   }
```

Gauss-Seidel: Let the Compiler do the work

- Specify the set of iterations that need to be executed
- Specify the dependencies that need to be obeyed.

```
#pragma omp for collapse(2) ordered(2)
for (i=1; i<N-1; i++)
  for (j=1; j<M-1; j++) {
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
    a[i][j] = 0.2 * (a[i-1][j] + a[i+1][j] +
                     a[i][j-1] + a[i][j+1] + a[i][j]);
    #pragma omp ordered depend(source)
  }
```

```
/* both loops collapsed, must obey ordering constraints */
#pragma omp for collapse(2) ordered(2)
for (i=1; i<N-1; i++)
  for (j=1; j<M-1; j++) {
    /* must wait until (i-1,j) and (i,j-1) iterations complete */
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)

    a[i][j] = 0.2 * (a[i-1][j] + a[i+1][j] +
                     a[i][j-1] + a[i][j+1] + a[i][j]);

    /* iteration (i,j) complete, let dependencies proceed */
    #pragma omp ordered depend(source)
  }
```

- Must likely inefficient because dependencies tracked at fine grain
- Can tile