CS420 – Lecture 7

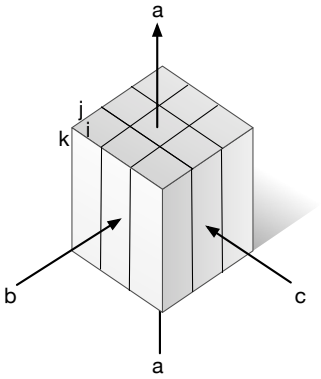
Marc Snir

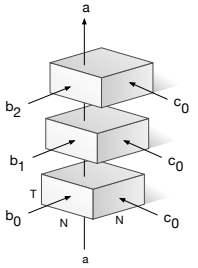
Fall 2018

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)





If tile in `k` dimension, need to add reduction.

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

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];
```

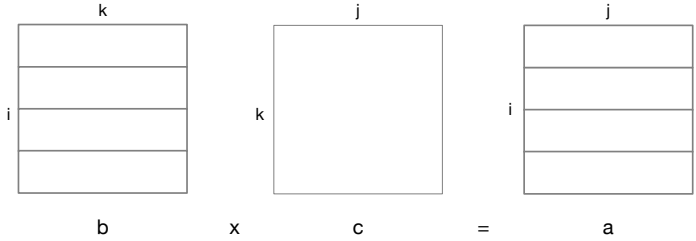
- The `reduction` clause takes an array section argument (will be discussed later)

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];
```

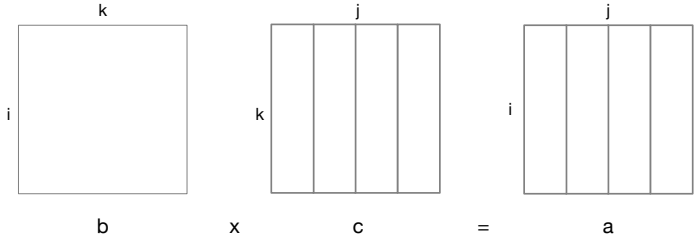
- `collapse(2)` indicates that two outermost loops should be taken as one loop (with N×N iterates) and executed in parallel

Different tiling = different partitions



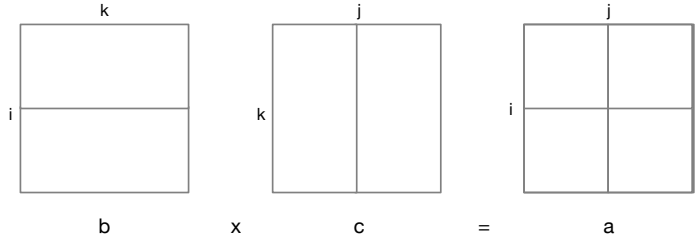
Tile `i`: Each thread computes product of horizontal tile of `b` with `c` that yields a horizontal tile of `a`.

Different tiling = different partitions



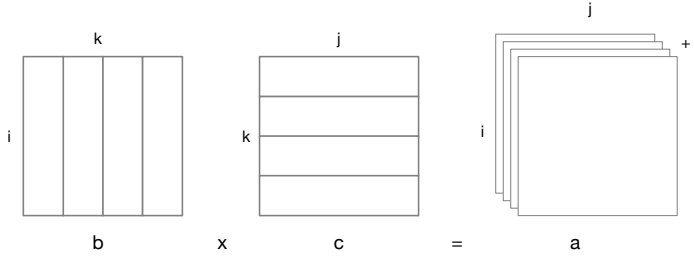
Tile `j`: Each thread computes product of `b` with vertical tile of `c` that yields a vertical tile of `a`.

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
Tile `k`	bus error

- Reduction on array sections is new feature – may not be well-supported
- Tiling choices impact locality

Sequential Jacobi

```
...
do {
  err = 0;
  k = 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);
...
```

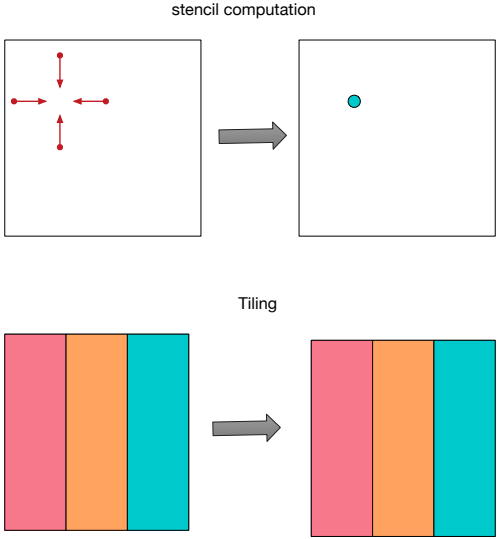
Parallel Jacobi

```
...
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);
...
```

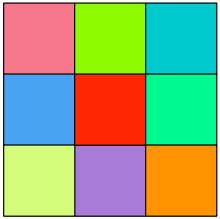
Only outer loop is executed in parallel

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

Assuming tiles are cache line aligned

Possibly better

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:err)
  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);
...
```

Solution 2: Use nested parallelism

```
...
do {
  err = 0;
  k = 1-k;
  #pragma omp parallel for reduction(max:err)
  for (jj=0; jj<n; jj++) {
    #pragma omp parallel for reduction(max:err)
    for (i=0; i<M; i++) {
      for (j=jj*T1+1; j<(jj+1)*T1+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);
...
```

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 id not obvious how many threads will execute a nested loop – could be one

Let's experiment

```
void report_num_threads(int level) {
  printf("Level %d: team size = %d\n",
         level, omp_get_num_threads());
}

int main(int argc, char *argv[]) {
  printf("available threads = %d\n",
         omp_get_max_threads());
  #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);
    }
  }
}
```

Executed:

```
export OMP_MAX_THREADS=7
./a.out
```

Output was:

```
available threads = 7
Level 1: team size = 2
Level 1: team size = 2
Level 2: team size = 1
Level 3: team size = 1
Level 2: team size = 1
Level 3: team size = 1
```

Nested parallelism

Use it at your own peril: Many implementations do not use "spare" threads to increase parallelism at lower levels
Note:

```
#pragma omp parallel for
```

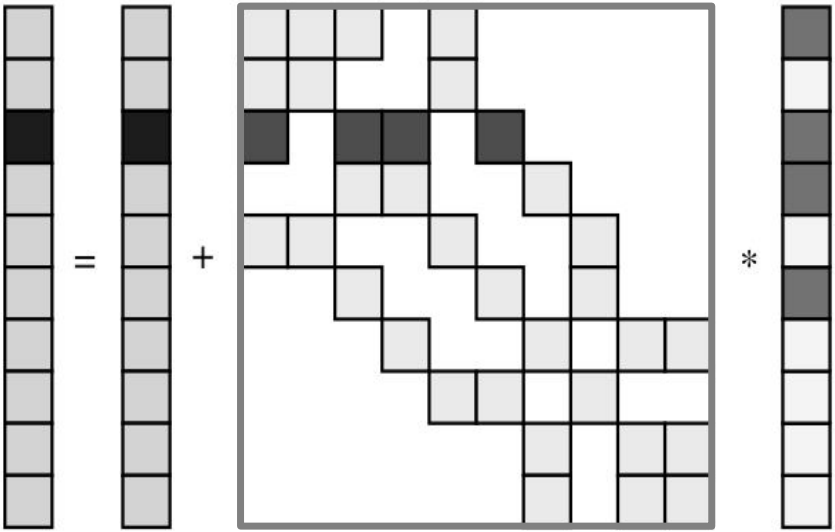
is equivalent to

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

first statement creates team; second statement does work sharing across team
Problem: Nested team creation might use only parent thread

Sparse data structures

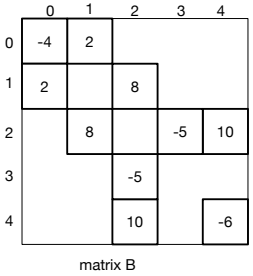
SparseMV: $a = b + Cd$ where C is a *sparse matrix*: most entries are zero.



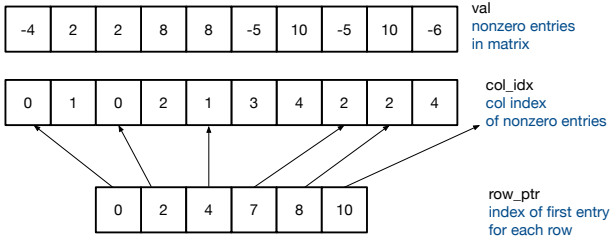
CRS

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

CRS: *Compressed Row Storage*

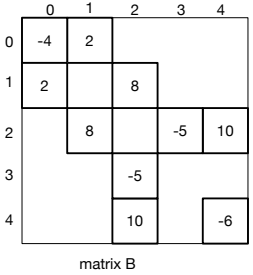


matrix B

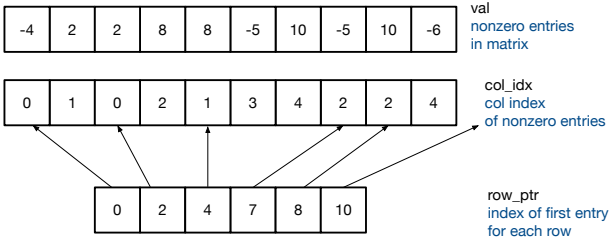


SparseMV

```
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]]
```



matrix B

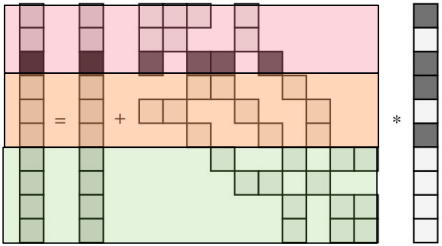


Parallel SparseMV

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.

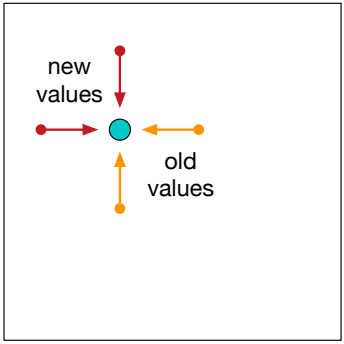
```
#pragma omp parallel for schedule(dynamic,T)
for (i=0; i<N; i++) {
  a[i] = b[i];
  for (j=row_ptr[i]; j<row_ptr[i+1]-1; j++)
    a[i] += val[j] * d[col_idx[j]];
}
```

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

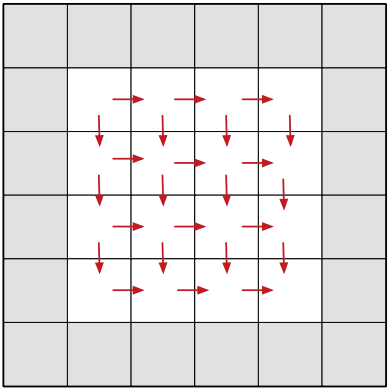
```
...
for (i=1; i<M-1; i++)
  for (j=1; j<N-1; j++)
    a[i][j] = 0.25 *
                (a[i-1][j] + a[i][j-1] +
                 a[i+1][j] + a[i][j+1]);
...
```

How do we parallelize?

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

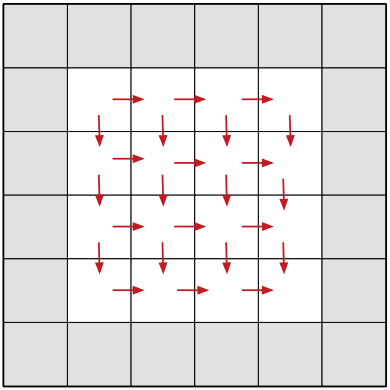
Wavefront

Loop carried dependencies

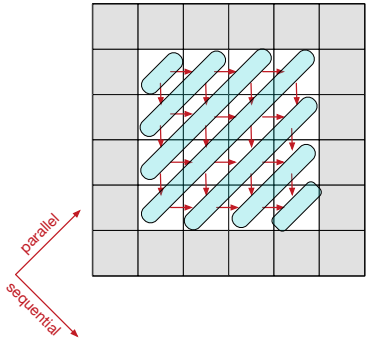


Wavefront

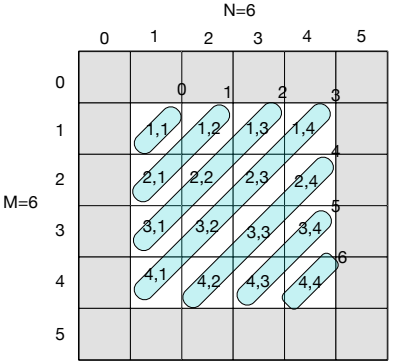
Loop carried dependencies



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) ? 1 : d-M+3;
  #pragma omp parallel for
  for (i=ifirst; i<=ilast; i++) {
    j = d+2-i;
    a[i][j] = 0.2 * (a[i][j]   +
          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)
  }
```

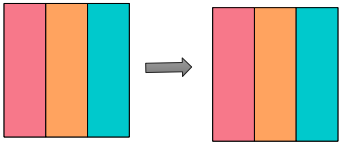
```
/* both loops collapsed, must obey ordering constraints */
#pragma omp for collapse(2) ordered(2)
for (i=1; i<N-1; i++)
  for (j=1; j<M-1; j++) {
    /* must wait until (i-1,j) and (i,j-1) iterations complete */
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)

    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

```
#pragma omp for collapse(2) ordered(2)
for (ii=1; ii<N-1; ii += T)
  for (jj=1; jj<M-1; j += T) {
    #pragma omp ordered depend(sink: ii-1,jj) depend(sink: ii,jj-1)
    for (i=ii; i<ii+T; i++)
      for (j=jj; j<jj+T; j++)
        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)
  }
```

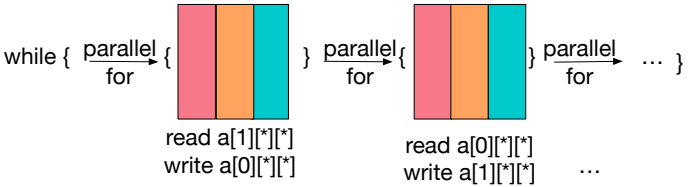
Back to Jacobi

```
...
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);
...
```



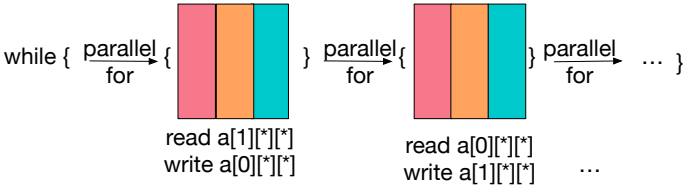
- Do we have races?

- 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

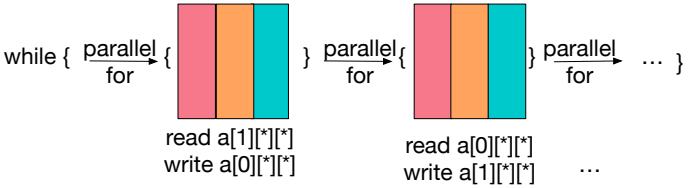


- Do we have communication between threads?

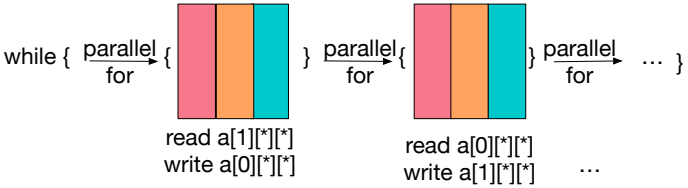
- Do we have communication between threads?
- Yes - "orange" thread needs a column written by each of "purple" and "blue" threads at previous "while" iteration, assuming threads pick same chunk at successive iterations.



- 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);
```

Metrics

Analyzing Parallel Performance

- Sequential performance $T_1(n)$: *time* to solve a problem of size n. (Same as $W(n)$, computation *work* needed to solve the problem.)
  - To simplify, assume (wrongly) each operation takes one time unit.
- Parallel performance $T_p(n)$: time to solve a problem of size n with p hardware threads
- Ideal world: Can run p time faster than with one hardware thread; $T_p(n) = T_1(n)/p$. World is not ideal.

Analyzing Parallel Performance

- Sequential performance $T_1(n)$: *time* to solve a problem of size n. (Same as $W(n)$, computation *work* needed to solve the problem.)
  - To simplify, assume (wrongly) each operation takes one time unit.
- Parallel performance $T_p(n)$: time to solve a problem of size n with p hardware threads
- Ideal world: Can run p time faster than with one hardware thread; $T_p(n) = T_1(n)/p$. World is not ideal.
  - Parallel code does more work than sequential code (e.g., spawning threads)

Analyzing Parallel Performance

- Sequential performance $T_1(n)$: *time* to solve a problem of size n. (Same as $W(n)$, computation *work* needed to solve the problem.)
  - To simplify, assume (wrongly) each operation takes one time unit.
- Parallel performance $T_p(n)$: time to solve a problem of size n with p hardware threads
- Ideal world: Can run p time faster than with one hardware thread; $T_p(n) = T_1(n)/p$. World is not ideal.
  - Parallel code does more work than sequential code (e.g., spawning threads)
  - Parallel code may not have enough parallelism – enough operations that can executed independently on distinct threads

Terminology

Speedup Ratio between sequential time and parallel time: $S_p(n) = T_1(n)/T_p(n)$.

Terminology

Speedup Ratio between sequential time and parallel time: $S_p(n) = T_1(n)/T_p(n)$.

- $S_1(n) = 1$

Terminology

Speedup Ratio between sequential time and parallel time: $S_p(n) = T_1(n)/T_p(n)$.

- $S_1(n) = 1$
- Normally $S_p(n) \leq p$; there may be cases of superlinear speedup: p caches have larger capacity than one cache.

Terminology

Speedup Ratio between sequential time and parallel time: $S_p(n) = T_1(n)/T_p(n)$.

- $S_1(n) = 1$
- Normally $S_p(n) \leq p$; there may be cases of superlinear speedup: p caches have larger capacity than one cache.
- Normally $S_p(n) \geq 1$; but it is not rare to find that a parallel program runs more slowly than a sequential program solving the same problem (parallel overheads and lack of parallelism)

Terminology

Speedup Ratio between sequential time and parallel time: $S_p(n) = T_1(n)/T_p(n)$.

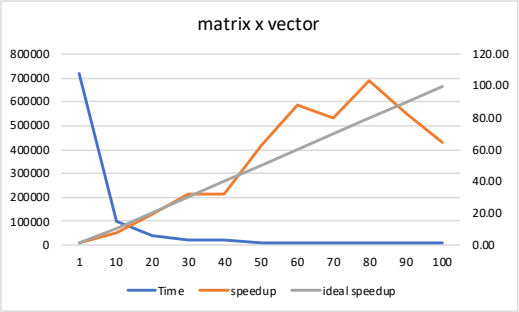
- $S_1(n) = 1$
- Normally $S_p(n) \leq p$; there may be cases of superlinear speedup: p caches have larger capacity than one cache.
- Normally $S_p(n) \geq 1$; but it is not rare to find that a parallel program runs more slowly than a sequential program solving the same problem (parallel overheads and lack of parallelism)

Efficiency Ratio between speedup and number of hardware threads: $E_p(n) = S_p(n)/p = T_1(n)/(p \times T_p(n))$; ratio between sequential work and parallel work. Normally $E_p(n) < 1$.

Example: Matrix×vector

```
#include <omp.h>
#include <time.h>
#include <stdio.h>
#define N 10000
double a[N][N], b[N], c[N];
int main(int argc, char *argv[]) {
  int n[] = {1, 10, 20, 30, 40, 50,
             60, 70, 80, 90, 100};
  int m;
  double time;
  for (m=0; m<11; m++) {
    omp_set_num_threads(n[m]);
    #pragma omp parallel
    {
      int i, j;
      time = omp_get_wtime();
      #pragma omp for collapse(2)
      for (i=0; i<N; i++)
        for (j=0; j<N; j++)
          c[i] += a[i][j] * b[j];
      printf("%d ",(int)(1000000.0 *
             (omp_get_wtime()-time)));
    }
    printf("\n");
  }
}
```

"Fake" experiment: measuring running time of each thread and taking maximum



- Execution time decreases as number of threads increase – But cannot decrease below the time to execute the inner loop (without code changes)
- Superlinear speedup in some regions

Amdahl's Law

Theorem

If a computation has a fraction α that can be executed in parallel and a fraction $1-\alpha$ that is sequential, then

$$S_p = \frac{1}{(1-\alpha)+\alpha/p}$$

Proof.

$$T_p = \frac{\alpha T_1}{p} + (1-\alpha)T_1$$

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\frac{\alpha T_1}{p} + (1-\alpha)T_1} == \frac{1}{(1-\alpha)+\alpha/p}$$

□

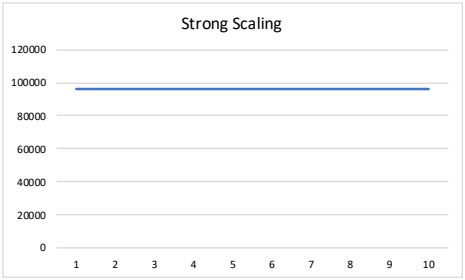
- Theorem holds as long as p does not exceed the level of available parallelism (N, in our example).
- Speedup can never exceed $\frac{1}{1-\alpha}$, the ratio between total work and sequential work.

More general formulation

- T_1 – number of operations
- T_∞ – longest critical path in the execution
- In our example $T_1(N) \sim N^2$ and $T_\infty(N) \sim N$
- Claim1: $T_p \geq \max(\frac{T_1}{p}, T_\infty)$
- Claim2: One can load-balance execution so that $T_p = O(\frac{T_1}{p} + T_\infty)$

Amdahl's Law revisited

- Amdahl's law was seen as evidence that parallelism has limited hope – eventually, the sequential part dominates.
- What was missing is a realization that larger machines are used to solve larger problems
- *Strong scaling*: Consider a fixed size problem and apply more and more processors to its solution.
  - Eventually, more processors do not help. (Too many cooks spoil the broth.)
- *Weak scaling*: Increase problem size as the number of processors is increased – keep work per processor constant.
  - Ideally, total compute time stays fixed



- Strong scaling of Matrix×vector
- Usually scaling is sublinear because of overheads of communication and synchronization

How do we study this?

- Study a function in two variables: $T_p(n)$ or $S_p(n)$
- Fix n and plot execution time $T_p(n)$ as function of n. This is *strong scaling*: Keep the work fixed and increase the number of workers; efficiency decreases
- Keep work per processor fixed and study increase in compute time. This is *weak scaling*
- Keep Efficiency fixed and study how problem size increases: $n_{1/2}(p) = \min\{n : \ E_p(n) \geq 0.5\}$

Speedup as function of n and p