CS420 – Lecture 13

Marc Snir

Fall 2018

Quiz 3

Choose all possible outputs to the following program. Assume exactly two threads are used.

```
int sum = 0;
int x = 0;
#pragma omp parallel   {
 #pragma omp single
  #pragma omp task depend(out: x)
   x = 2;
 #pragma omp single nowait
  #pragma omp task depend(in: x)   {
   int y = 3*x;
   #pragma omp atomic write
    sum = y;
  }
  #pragma omp single nowait
   #pragma omp task depend(in: x)   {
    int y = 5*x;
  #pragma omp atomic write
    sum = y;
  }
 #pragma omp taskwait
}
printf("%d\n", sum);
```

- `nowait`: other threads do not have to wait for the completion of the statement
- First task will execute first; second and third may complete out of order.
- Third task executes the assignment to `sum` last: 10 printed
- Second task executed the assignment to `sum` last: 6 printed.

Consider two MPI processes, P1 and P2. Answer on the likelihood of the following program deadlocking:

P1:	P2:
MPI_Send(P2)	MPI_Send(P1)
MPI_Recv(P2)	MPI_Recv(P1)

Answer

- Never deadlock
- May deadlock
- Always deadlock

Consider two MPI processes, P1 and P2. Answer on the likelihood of the following program deadlocking:

P1:	P2:
MPI_Send(P2)	MPI_Send(P1)
MPI_Recv(P2)	MPI_Recv(P1)

Answer

- Never deadlock
- May deadlock
- Always deadlock

```
 #define SIZE 12

int main(int argc, char *argv[]) {
 int rank;
 int myResult = 0, result;
 float sendbuf[SIZE] = {10,12,8,6,3,9,16,9,5,8,5,10};
 float recvbuf[3];
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 MPI_Scatter(sendbuf, 3, MPI_FLOAT, recvbuf, 3, MPI_FLOAT, 0,
        MPI_COMM_WORLD);

 for (int i = 0; i < SIZE / 4; i++)
    myResult += recvbuf[i];

 MPI_Reduce(&myResult, &result, SIZE, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

 if (0 == rank) {
  printf("The sum is %d.\n", result);
  }
 MPI_Finalize();
}
```

What is the outout of mpirun -np 1 a.out

What is the outout of mpirun -np 1 a.out
Only one process is executing code. Process 0 will send to itself the first 3 elements of `sendbuf`, and the reduction copies myresult to result
The sum is $10 + 12 + 8 = 30$.

What is the outout of mpirun -np 1 a.out
Only one process is executing code. Process 0 will send to itself the first 3 elements of `sendbuf`, and the reduction copies myresult to result
The sum is $10 + 12 + 8 = 30$.

Based on the aforementioned code in Question 3, what is the output of mpirun -np 4 a.out?

What is the outout of mpirun -np 1 a.out
Only one process is executing code. Process 0 will send to itself the first 3 elements of `sendbuf`, and the reduction copies myresult to result
The sum is $10 + 12 + 8 = 30$.

Based on the aforementioned code in Question 3, what is the output of mpirun -np 4 a.out?
Four processes participate. Process 0 is sending 3 elements to each; they compute, respectively, $10 + 12 + 8 = 30$, $6 + 3 + 9 = 18$, $16 + 9 + 5 = 30$, and $8 + 5 + 10 = 23$. The reduce sums these four numbers ant sets `sum` to $30 + 18 + 30 + 23 = 101$.

Consider two MPI processes, P1, P2, and P3. Answer on the likelihood of the following program deadlocking:

P1:	P2:	P3:
MPI_Send(P2)	MPI_Send(P3)	MPI_Recv(P2)
MPI_Recv(P3)	MPI_Recv(P1)	MPI_Send(P1)

Answer

- Never deadlock
- May deadlock
- Always deadlock

Consider two MPI processes, P1, P2, and P3. Answer on the likelihood of the following program deadlocking:

P1:	P2:	P3:
MPI_Send(P2) MPI_Recv(P3)	MPI_Send(P3) MPI_Recv(P1)	MPI_Recv(P2) MPI_Send(P1)

Answer

- Never deadlock
- May deadlock
- Always deadlock

Mid-Term

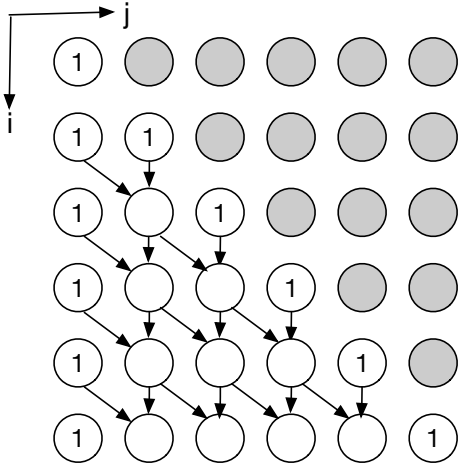
Exam-Q3

```
#include <omp.h>
#define N 1000
int a[N][N];

int main() {
  int i,j;
  for(i=0;i<N;i++)
  a[i][0]=a[i][i]=1;
  for(i=1;i<N;i++)
  for(j=1;j<i;j++)
  a[i][j]=a[i-1][j-1]+a[i-1][j];
}
```

1	0	0	0	0	0
1	1	0	0	0	0
1	2	1	0	0	0
1	3	3	1	0	0
1	4	6	4	1	0
1	5	10	10	5	1

Binomial coefficients

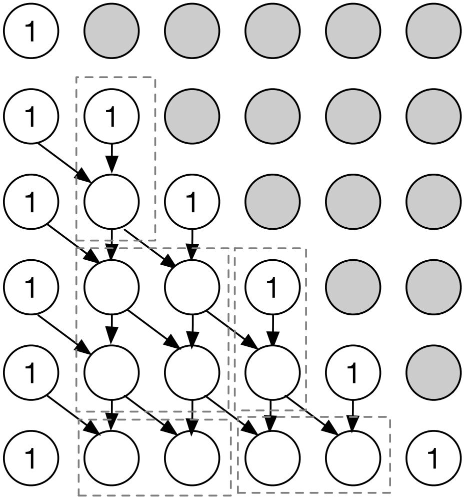
Dependencies



Can parallelize in j dimension, but not in i dimension – can parallise only inner loop.

Tiled code

```
for(ii=1; ii<N; ii+=T) {
  iinext = ii+T<N?ii+T:N;
  for(jj=1; jj<iinext; jj+=T)
  for(i=ii; i<iinext; i++) {
    jnext = jj+T<i?jj+T:i;
    for(j=jj; j<jnext; j++)
    a[i][j]=a[i-1][j-1]+a[i-1][j];
  }
}
```

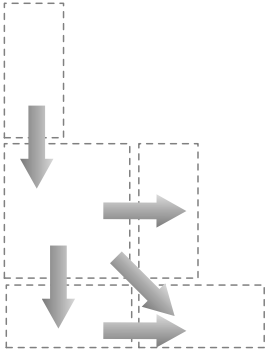


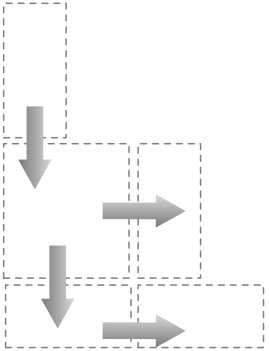
Any improved sequential performance from tiling?

- Tiling is advantageous if it reduces cache misses, by improving temporal locality – reducing reuse distance.
- Each entry of array `a` is accessed three times = one write and two reads (`a[i][j]=a[i-1][j-1]+a[i-1][j]`)
- About `2N` accesses to distinct entries are separating the first and second access to `a[i][j]`, and one other access spearate the second and third access.
- If `N` is much smaller than the cache size, then only the first access can be a cache miss.
- So temporal locality is not improved
- Nor is spatial locality improved – tiling cannot improve it
- Tiled code is likely to perform worse

Parallel Code

- Will have each tile computed by a separate task.
- Need to enforce dependencies: Before tile is computed, need to complete computing tile to the left, above and diagonally left above
- Sufficient to enforce first two dependencies (left and above).





- It does not matter which variables we pick to enforce the dependencies, as long as the "in-out" pairs enforce the right order, and the references are valid.
- Can pick `depend(out: a[ii][jj])`, `depend(in: a[ii-1][jj], a[ii][jj-1])`

code

```
#pragma omp parallel /* need to start team */
 #pragma omp single  /* need to start all tasks on one thread, */
         /* so that they are generated once, in the right order  */
  for(ii=1; ii<N; ii+=T) {
   iinext = ii+T<N?ii+T:N;
   for(jj=1; jj<iinext; jj+=T)
   #pragma omp task depend(out: a[ii][jj]) \
      depend(in: a[[ii-1][jj],a[ii][jj-1]) \
      firstprivate(ii,jj) /* Need private copies of ii and jj */
    for(i=ii; i<iinext; i++) {
     jnext = jj+T<i?jj+T:i;
     for(j=jj; j<jnext; j++)
      a[i][j]=a[i-1][j-1]+a[i-1][j];
     }
  }
```

Matrix Product

Communication time – Matrix product

Broadcast of n items: time $\approx (\lg p)(\ell + n/b)$

Reduce of n items ibid

Scatter of n items time $\approx (\lg p)\ell + n/b$

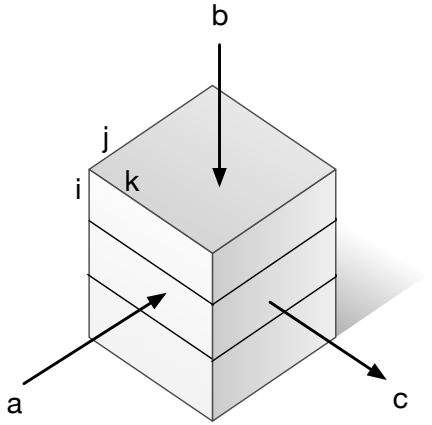
Gather of n items ibid

Note: All versions have same computation time

Assume all matrices are $N \times N$ ande are initially on process 0.Result is returned to process 0.

1D tiling

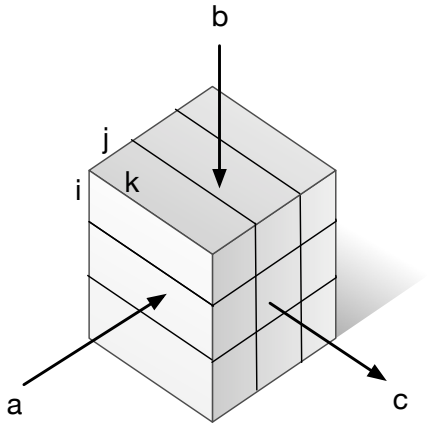
- Scatter a: $t \approx \ell \lg p + N^2/b$
- Broadcast b: $t \approx \ell \lg p + N^2 \lg p/b$
- Gather c: $t \approx \ell \lg p + N^2/b$
- Total: $t = \Theta(\lg p(\ell + N^2/b)$
  Communication time dominated by broadcast of b.



2D tiling

a and b are partitioned into $\sqrt{p}$ tiles; c is partitioned into p tiles.

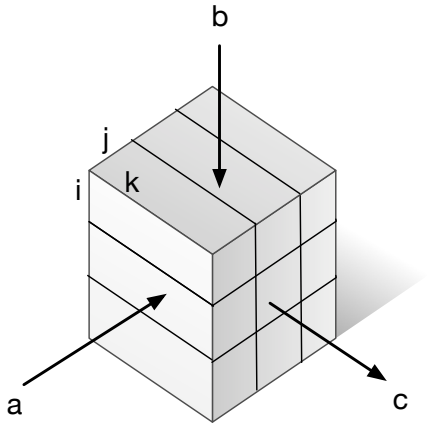
- Scatter a to $\sqrt{p}$ processes, next broadcast each tile to $\sqrt{p}$ processes:
  $t \approx (0.5\ell \lg p + N^2/b) + (0.5 \lg p(\ell + N^2/(b\sqrt{p}) = \approx \ell \lg p + N^2/b$
- Scatter and broadcast b: ibid
- Gather c: $t \approx \ell \lg p + N^2/b$
- Total: $t = \approx \ell \lg p + N^2/b$



3D Tiling

a, b and c are partitioned into $p^{2/3}$ tiles.

- Scatter a to $p^{2/3}$ processes, next broadcast each tile to $p^{1/3}$ processes: $t \approx \frac{2}{3}\ell \lg p + N^2/b) + (\frac{1}{3} \lg p(\ell + N^2/(bp^{2/3}) \approx \ell \lg p + N^2/b$
- Scatter and broadcast b: ibid
- Reduce c: $p^{2/3}$ parallel reductions of a vector of length $n^2/p^{2/3}$ over $p^{1/3}$ processes: $t \approx \lg p(\ell + N^2/(bp^{2/3}))$
- Total: $t \approx \ell \lg p + N^2/b$



- 1D tiling: $t \approx \ell \lg p(N)^2/b \lg p$
- 2D tiling $t \approx \ell \lg p + N^2/b$
- 3D tiling $t \approx \ell \lg p + N^2/b$
- 3D tiling only changes constants; communication time is dominated by scatter time.

Assume that matrices are already distributed (over p, $p^{1/2}$ or $p^{2/3}$ processes, respectively)

- 1D tiling: $t \approx \ell \lg p + N^2/b$
- 2D tiling: $t \approx \ell \lg p + N^2/(bp^{1/2})$
- 3D tiling: $t \approx \ell \lg p + N^2/(bp^{2/3})$

One-Sided Communication

One sided

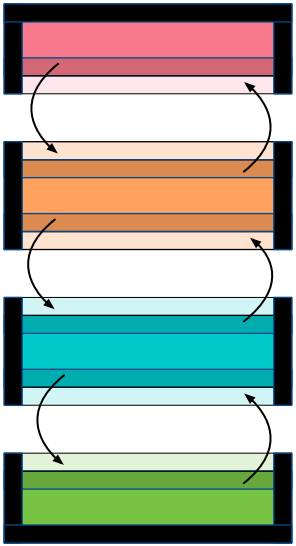
One cause of inneficiency in MPI is the overhead of matching sends to receives.

- Comparing communicator, source and tag
- Handling wildcard source and wildcard tag
- Matching message in the right order

In may cases, this is superflous, since sender "knows" were the receive buffer is (same receive buffer used again and again)
One sided communication can be used – only one side (sender or receiver) needs to call MPI.
put/get/accumulate

Jacobi- 1D tiling

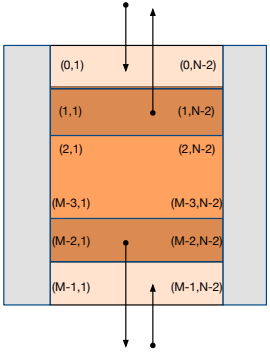
```
...
...
double a[2][M][N];
...
MPI_Isend(&a[1][1][1],N-2,MPI_DOUBLE,
    rank-1,0,MPI_COMM_WORLD,&req[0]);
MPI_Irecv(&a[1][0][1],N-2,MPI_DOUBLE,
   rank-1,0,MPI_COMM_WORLD,&req[1]);
...
MPI_Isend(&a[1][M-2][1],N-2,MPI_DOUBLE,
    rank+1,0,MPI_COMM_WORLD,&req[2]);
MPI_Irecv(&a[1][M-1][1],N-2,MPI_DOUBLE,
    rank+1,0,MPI_COMM_WORLD,&req[3]);
...
```



With 1-sided communication

```
...
MPI_Win win; /* declare window */

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

/* expose array to remote accesses */
MPI_Win_create(a, sizeof(a), sizeof(a[0]), NULL,
     MPI_COMM_WORLD, &win);

for(iter=0; iter<MAX; iter++) {
/* compute first and last row */
 for(j=1; j<N-1; j++)
  a[1-k][1][j] = 0.25*(a[k][0][j]+a[k][2][j]
     +a[k][1][j-1]+a[k][1][j+1]);
 for(j=1; j<N-1; j++)
  a[1-k][M-2][j] = 0.25*(a[k][M-3][j]+a[k][M-1][j]
     +a[k][M-2][j-1]+a[k][M-2][j+1]);
```

```
/* start communication */
 MPI_Win_fence(0, win);

 if (rank>0)  /* send data to neighbor above */
  MPI_Put(&a[k][1][1], N-2, MPI_DOUBLE, rank-1,
      (int)(&a[1-k][M-1][1] -a), N-2, MPI_DOUBLE, win);

 if (rank<size-1) /* send data to neighbor below */
  MPI_Put(&a[k][M-2][1], N-2, MPI_DOUBLE, rank+1,
     (int)(&a[1-k][0][1]-a), N-2, MPI_DOUBLE, win);
/* compute interior */
 for(i=1;i<M-2;i++)
  for(j=1;j<N-2;j++)
   a[1-k][i][j] = 0.25*(a[j][i-1][j]+a[k][i+1][j]
     +a[k][i][j-1]+a[k][i][j+1]);
 MPI_Win_fence(0, win) /* complete communication */
 k=1-k;
```



```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

base: starting address of window

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

base: starting address of window

size: size of window in bytes

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

base: starting address of window

size: size of window in bytes

disp_unit: local unit size for displacements, in bytes

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument
- comm: processes involved

```
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument
- comm: processes involved
- win: window object returned by the call

`MPI_Win_create(base, size, disp_unit, info, comm, win)`

- base: starting address of window
- size: size of window in bytes
- disp_unit: local unit size for displacements, in bytes
- info: info argument
- comm: processes involved
- win: window object returned by the call

Also

- `MPI_Win_allocate` – allocate new memory when window is created; and
- `MPI_Win_create_dynamic` and `MPI_win_attach` – create zero size window and dynamicall attach new memory as needed.

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

target_disp: displacement from start of window to remote buffer

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

target_disp: displacement from start of window to remote buffer

target_count: number of entries in remote buffer

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

target_disp: displacement from start of window to remote buffer

target_count: number of entries in remote buffer

target_datatype: type of each entry in remote buffer

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
target_disp, target_count, target_datatype, win)
```

origin_addr: starting address of local buffer

origin_count: number of entries to put

origin_datatype: type of each entry in local buffer

target_rank: rank of remote process

target_disp: displacement from start of window to remote buffer

target_count: number of entries in remote buffer

target_datatype: type of each entry in remote buffer

win: window used for communication

```
MPI_WIN_FENCE(assert, win)
```

`MPI_WIN_FENCE(assert, win)`

assert information on type of communication

`MPI_WIN_FENCE(assert, win)`

assert information on type of communication

win window

`MPI_WIN_FENCE(assert, win)`

assert information on type of communication

win window

Acts like a barrier: starts and complete a *communication epoch*

Example: Distributed table

- Table too large to fit on one process – it is distributed across all processes
- assume integer indices and double values
- Three operations:
  - `double read_table(int index)`
  - `void write_table(int index, double value)`
  - `void increment_table(int index, double increment)`
- Accesses to table are not bulk synchrounous – each process can access the distributed table independently
- Need to ensure atomicity of accesses (mutual exclusion)

implementation

```
...
#define TABLE_SIZE 10000000000

int local_size; /* local table size */

main() {
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int local_size = (TABLE_SIZE+size-1)/size;
double table[local_size];
MPI_Win_create(table, sizeof(table), sizeof(table[0]), 0,
     MPI_COMM_WORLD, win);
```

Cannot use fence, since each process can call the 3 functions at any point in time.

```
double read_table(int index) {
double value;
int target_rank= index%size;
int  target_disp= index/size;
MPI_Win_lock(MPI_LOCK_SHARED, target_rank, 0, win);
MPI_Get(&value, 1, MPI_DOUBLE, target_rank, target_disp, 1, MPI_DOUBLE, win);
MPI_Win_unlock(target_rank, win);
return value;

void write_table(int index, double value) {
int target_rank= index%size;
int  target_disp= index/size;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target_rank, 0, win):
MPI_Win_put(&value, 1, MPI_DOUBLE, target_rank, target_disp, 1, MPI_DOUBLE,
    win);
MPI_Win_unlock(rank, win);
}
```

```
void increment_table(int index, double value) {
int rank= index%rank;
int  local_index= index/rank;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, 0, win):
MPI_Accumulate(&value, 1, MPI_DOUBLE, rank, local_index, 1, MPI_DOUBLE,
    MPI_SUM, win);
MPI_Win_unlock(rank, win);
}
```

- `MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)` – same arguments as `MPI_Put`.
- `MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)` – same arguments as `MPI_Put`, with the addition of the operation argument.
- `MPI_Win_lock(lock_type, rank, assertion, win)`
- `MPI_Win_unlock(rank, win)`