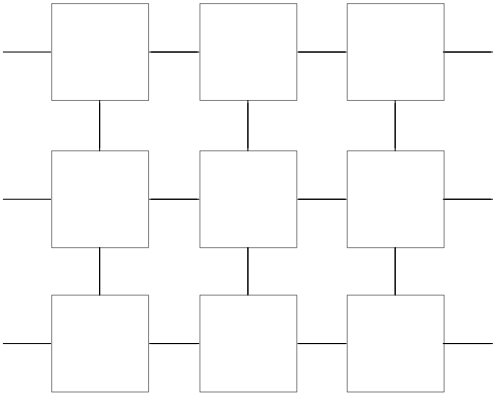
CS420 – Lecture 16

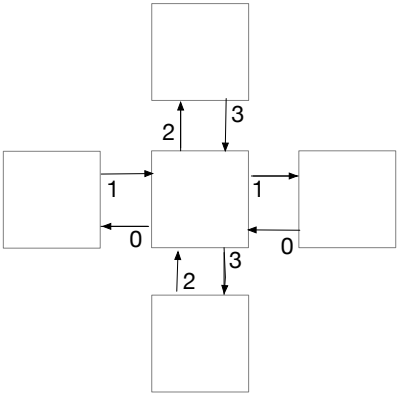
Marc Snir

Fall 2018

Sparse collectives explained again

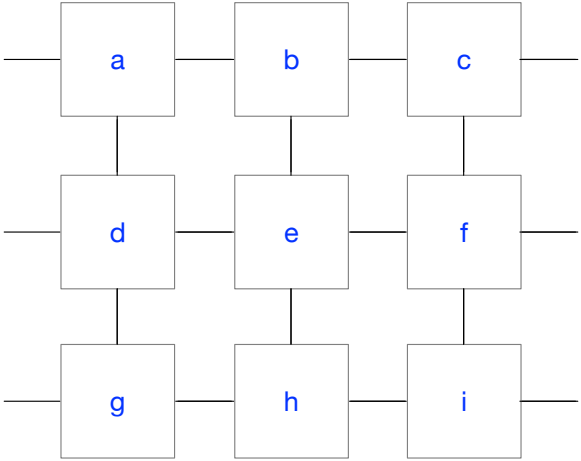
Cartesian topology is also a graph topology: Each node has 2D incoming and 2D outgoing edges (with possible exception of boundaries).





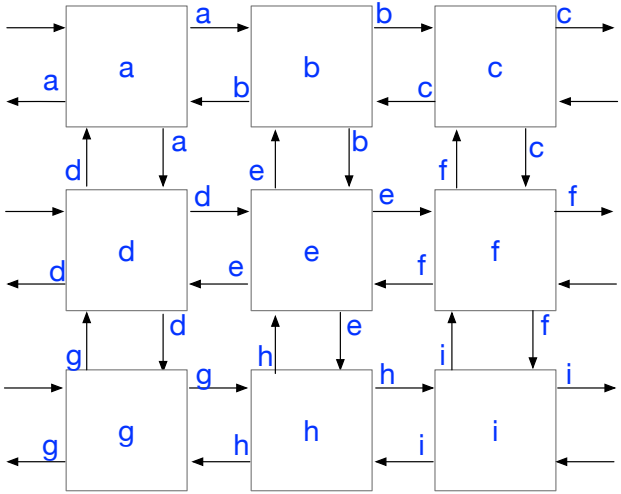
MPI_Neighbor_allgather

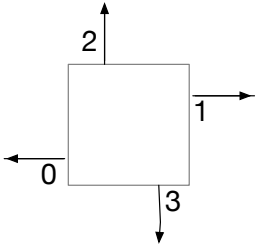
```
MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)
```



MPI_Neighbor_allgather – communication

```
MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)
```

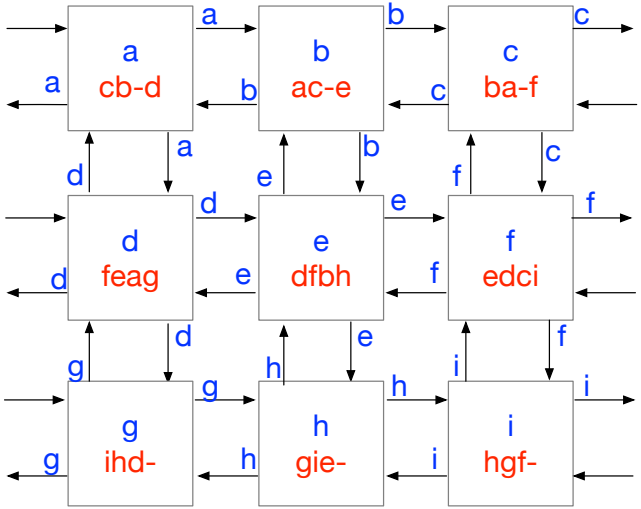


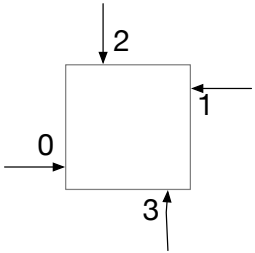


MPI_Neighbor_allgather – result

`MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)`

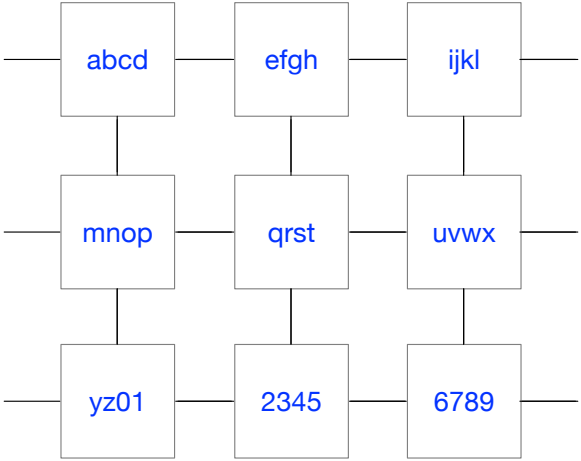
Dash (-) indicates locations that is not updated by call





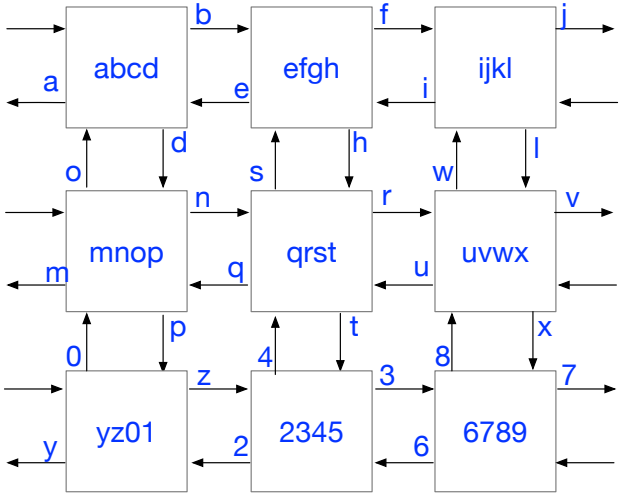
MPI_Neighbor_alltoall

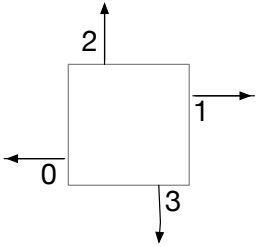
```
MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)
```



MPI_Neighbor_allgather – communication

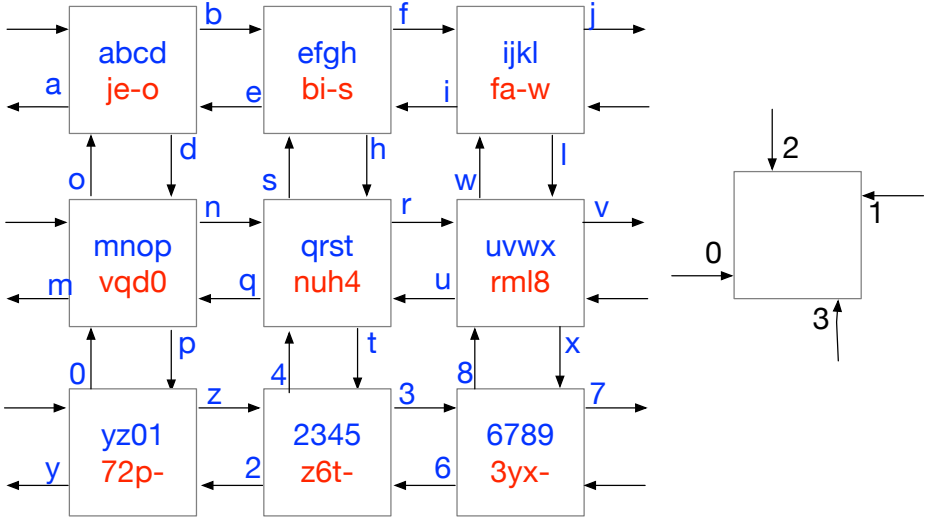
```
MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)
```





MPI_Neighbor_allgather – result

```
MPI_Neighbor_allgather(sendbuf, 1, MPI_CHAR, recvbuf, 1, MPI_CHAR, comm)
```



Variants

The two calls have a "v" variant where communicated messages can have diffferent lengths:
`MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)`
`MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)`
Each of the four calls have a nonblocking variant.

Back to GPU

Simple example

```
void vec_mult(int N, float p[N], float v1[N], float v2[N]) {
int i;
#pragma omp target
#pragma omp parallel for private(i)
for (i=0; i<N; i++)
p[i] = v1[i] * v2[i];
}
```

- `pragma omp target`: the block of code following the pragma is a task that will execute on the default *target* (default GPU)
- `private` indicates that the executing task will get a private copy of `i`.
- By default, the variables `p`, `v1` and `v2` are copied to GPU memory when the task starts and copied back when the task end. (We shall see late how to avoid superfluous copies.)

More precisely "variables are mapped for the extent of the region from the data environment of the encountering task, to the device data environment."
Translation to plain English:

- During the execution of the region, the variables are "owned" by the target device (GPU) – should not be accessed by the CPU.
- If GPU cannot access CPU memory, then data will be copied from CPU memory to GPU memory and back.
- If GPU can access CPU memory, compiler may decide not to copy

Improved code

```
void vec_mult(int N, float p[N], float v1[N], float v2[N]) {
 int i;
 #pragma omp target map(to: N, v1, v2) map(from: p)
   #pragma omp parallel for private(i)
   for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
 }
```

- `map(to:)`: variables are copied in when kernel starts, but not copied out when it ends.
- `map(from:)`: variables are copied out when kernel completes, but not copied in when it starts.
- `map(alloc:)` allocate in GPU memory
- `map(tofrom:)` copy in at start, out at finish (default)
- `map(release:)` free the allocated GPU memory[exit call]
- `map(delete:)` delete the allocated memory [exit call]

What if kernel does not need entire vector?

Can copy an array section

```
void vec_mult(int N, int first, int last, float p[N], float v1[N], float v2[N])
{
 int i;
 int length = last-first+1;
 #pragma omp target map(to: first, last, v1[first:length], v2[first:length]) \
           map(from: p[first:length])
   #pragma omp parallel for private(i)
  for (i=first; i<=last; i++)
     p[i] = v1[i] * v2[i];
}
```

if p is declared as `int p[n]` then

- `p[first:length]` = `p[first]`, `p[first+1]`,...,`p[first+length-1]`
- `p[first:]` = `p[first]`, `p[first+1]`, ..., `p[N-1]`
- `p[:length]` = `p[0]`, `p[1]`,...,`p[length-1]`

Can have array sections of multidimensional arrays

```
float x[5][7];
#pragma omp target map(x[2:2][5:2])
...
```

The array elements `x[2][5]`, `x[2][6]`, `x[3][5]`, `x[3][6]` will be copied back and forth

Can we leave data in GPU memory across kernel invocations?

```
...
#pragma omp target data map(from: N, p)
{
 #pragma omp target map(to: v1, v2)
  #pragma omp parallel for private(i)
  for (i=0; i<N; i++)
   p[i] = v1[i] * v2[i];
 recompute(N, v1[N], v2[N]);
 #pragma omp target map(to: v1, v2)
  #pragma omp parallel for private(i)
  for (i=0; i<N; i++)
   p[i] = v1[i] * v2[i];
}
```

- `p` and `N` stay in GPU memory until the end of the `target data map` block
- `v1` and `v2` are moved into GPU memory twice, for each instance of the `target map` block

Jacobi – yet again

Sequential code

```
  ...
while(error>tol) {
 error = 0.0;  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][j-1] +  a[k][i][j+1] + a[k][i-1][j]
      + a[k][i+1][j]);
   error = fmax ( error, fabs (a[k][i][j] - a[1-k][i][j]));
   }
 }
```

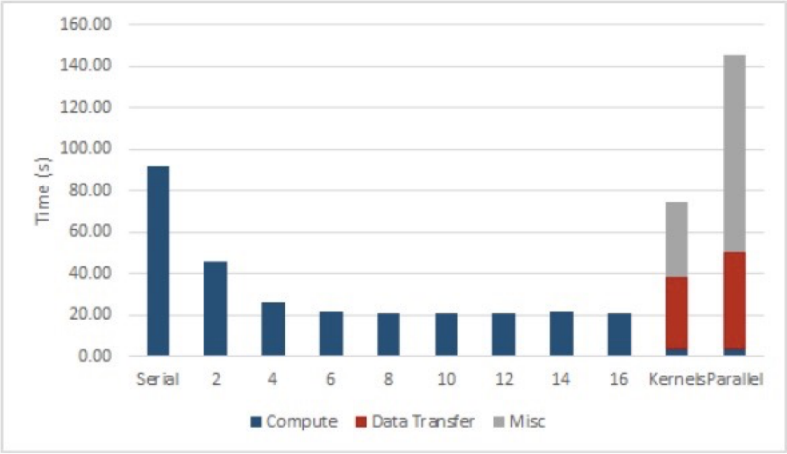
Execute on GPU (and trust the compiler to parallelize code)

```
...
while(error>tol) {
 error = 0.0;   k=1-k;
 #pragma omp target
 for (i = 1;   i < m-1;   i++ )
  for (j = 1; j < n-1; j++ )  {
   a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
    + a[k][i+1][j]);
   error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
   }
}
```

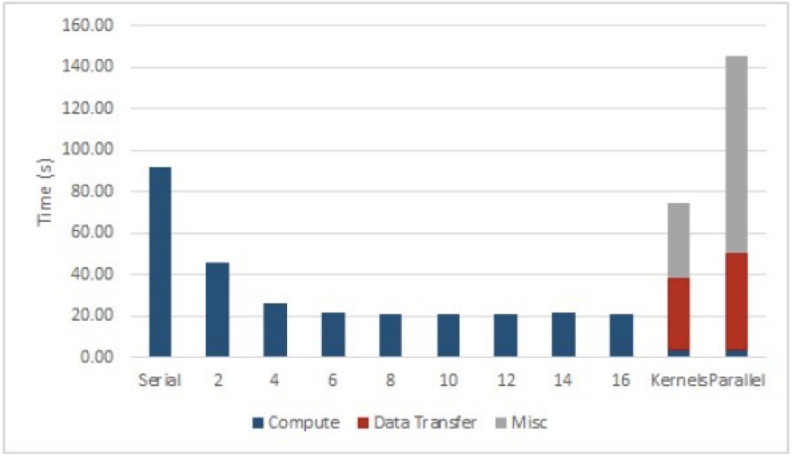
Execute on GPU (and specify parallelism)

```
...
while(error>tol) {
 error = 0.0;  k=1-k;
 #pragma omp target
 #pragma omp parallel for collapse(2) reduction(max:error)
  for (i = 1;  i < m-1;  i++ )
   for ( j = 1; j < n-1; j++ )  {
   a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
     + a[k][i+1][j]);
   error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
  }
 }
```

Performance

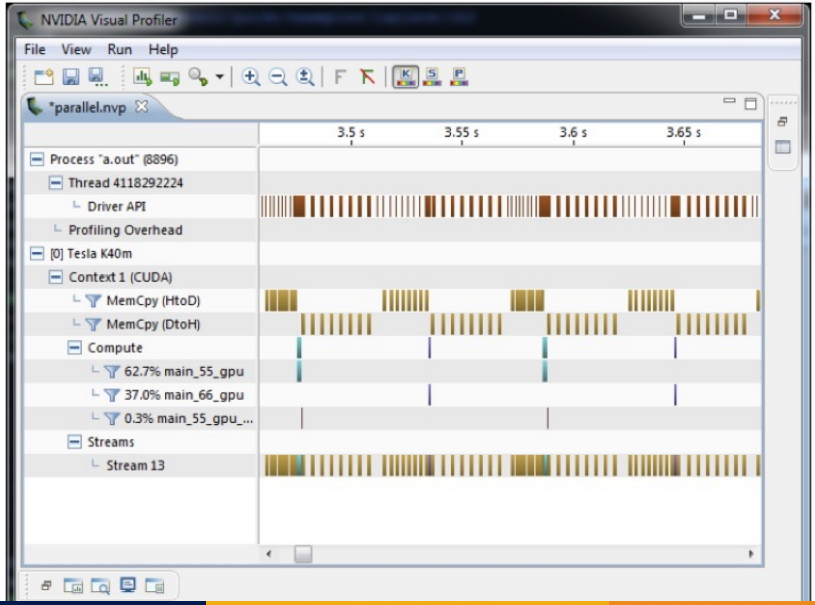


Performance



At each iteration of the while loop the arrays are needlessly copied btw CPU and GPU

Use performance tools to find bottlenecks



Avoid needless copying

```
...
#pragma omp target data map(a,m,n)
while (error > tol)  {
error = 0.0;  k=1-k;
#pragma omp target map(error)
#pragma parallel loop collapse(2) reduction(max:error)
for (int i = 1;  i < m-1;  i++ )
for (int  j = 1; j < n-1; j++ )  {
  a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
  + a[k][i+1][j]);
  error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
}
```

Can do the convergence test on the GPU

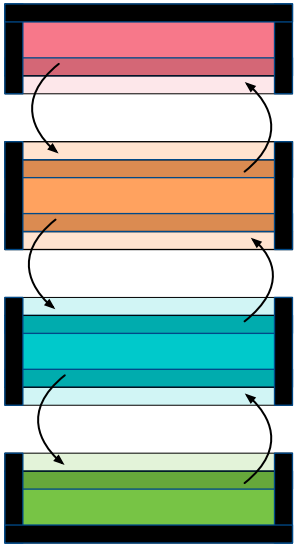
```
...
#pragma omp target map(a,m,n,error)
while (error > tol)  {
error = 0.0; k=1-k;
#pragma parallel loop collapse(2) reduction(max:error)
for (int i = 1;  i < m-1;  i++ )
for (int  j = 1; j < n-1; j++ )  {
  a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
  + a[k][i+1][j]);
  error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
}
```

- Using the GPU is not worthwhile if the kernel is too small – better to execute on CPU
- The `target` pragma can have an `if` clause

```
...
#pragma omp target if (n> minlen) map(a,m,n,error)
while (error > tol)  {
error = 0.0; k=1-k;
#pragma parallel loop collapse(2) reduction(max:error)
for (int i = 1;  i < m-1;  i++ )
for (int  j = 1; j < n-1; j++ )  {
  a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
  + a[k][i+1][j]);
  error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
}
```

- Parallel loop will execute on GPU if condition is statisfied, CPU, otherwise.
- Other OpenMP pragmas (e.g., `parallel loop`) can also have `if` clauses.

Jacobi: GPU + MPI



- Normally, MPI executes on CPU and communication buffers should be in CPU memory
- $\Rightarrow$ Need to copy boundary rows from GPU to CPU and ghost rows from CPU to GPU

```
...
#pragma omp target map(a,m,n,error)
while (error > tol)  {
error = 0.0;  k=1-k;
/* compute top and bottom rows */
#pragma omp target
#pragma omp parallel for
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]);
a[1-k][m-2][j] =
0.25*(a[k][m-3][j]+a[k][m-1][j]+a[k][m-1][j-1]+a[k][m-1][j+1]);
#pragma omp target update from(a[1-k][1][1:n-2], a[1-k][m-2][1:n-2]
```

`target update to(`*list*`), from(`*list*`)` CPU copy is updated from GPU copy, or vice versa

```
/* start communication */
if (myrank>0) {
MPI_Isend(&a[1-k][1][1], n-2, MPI_DOUBLE, myrank-1, 0, &req[0]);
MPI_Irev(&a[1-k][0][1], n-2, MPI_DOUBLE, myrank-1,0,&req[1]);
}
if(myrank<size-1) {
MPI_Isend(&a[1-k][m-2][1], n-2, MPI_DOUBLE, myrank+1, 0, &req[2]);
MPI_Irev(&a[1-k][m-1][1], n-2, MPI_DOUBLE, myrank+1,0,&req[3]);
}
/* compute interior on GPU */
#pragma target
#pragma parallel loop collapse(2) reduction(max:error)
for (int i = 2;  i < n-2;  i++ )
for (int  j = 1; j < m-1; j++ )  {
a[1-k][i][j] = 0.25 * ( a[k][i][j-1] +  a[k][i][j+1] + a[k][i-1][j]
+ a[k][i+1][j]);
error = fmax ( error, fabs (a[k][i][j] -a[1-k][i][j]));
}
```

```
/* end communication */
if (myrank == 0)
MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
else if (myrank == size-1)
MPI_Waitall(2, &req[2], MPI_STATUSES_IGNORE);
else
MPI_Waitall(4, req, MPI_STATUSES_IGNORE);
/* update gpu memory */
#pragma omp target update
to(a[1-k][0][1:n-2], a[1-k][m-1][n-2]) from(error)
}
```

How about calling functions on target device?

Need to tell compiler that the function has to be compiled for GPU

```
#pragma omp declare target
void vect_mult(int N, float v1[N], float v2[N], float p[N]) {
#pragma omp parallel for
for(int i=0; i<N; i++)
p[i] = v1[i] * v2[i];
}
#pragma omp end declare target

int main() {
...
#pragma omp target if (m>1000)
vect_mult(m, x, y, z);
}
```

Compiler compiled both GPU and CPU versions