Tu. 05/08/12:

HW 4 — some comments on report:

For tables and figures, you should think of writing in 4 steps:

- Introduce each: "Table 3.1 shows the results of a serial run of ---."
- Define the contents: "It shows timing data in seconds for 1, 2, 4, 8 nodes using 1, 2, 4, or 8 process per node."
- Describe the contents: "As we double the number of process, the times ...."
- Draw conclusions or make comparison: "this shows the scalability of the algorithm/code/implementation/hardware."

Memory prediction:

You can focus on large arrays immediately, here only $l\_A \in \mathbb{R}^{n \times l\_n}$. But also, can just do it for 1-process code! Reason: We want to use this to determine which n to use => So, need to know if it fits on 1 node! If it fits there, it will fit on any number of nodes.

We want $n = 2^v$. Ex.: $n = 8192 = 2^{13}$

memory for A = 8 B/double * n^2 double = $8n^2$ B

$= 8 \cdot (2^{13})^2 B = 2^3 2^{26} = 2^{29} B = 2^{19}$ kB $= 2^9$ MB

$= 0.5 \cdot 2^{10}$ MB = 0.5 GB

N = 8192 => M = 0.5 GB

N = 16k => M = 2 GB

N = 32k => M = 8 GB

HW5: Specify the hardware used and software versions.

→ Put this at end of Sec. 1.

Starting point for HW6 is HW5 = serial C code for Ax() — all other parts of the code should already be parallel!

Step 0 : Restate serial code, but with l_u instead of u, l_v for v, and l_j for j

(For debugging, have no "j" defined, so that you can be sure that all j are now l_j.)

```
for ( l_j = 0 ; l_j < l_N ; l_j++)
    for ( i = 0 ; i < N ; i++ ) {
            tmp = 4.0 * l_u [i + N * l_j]
            if ( l_j > 0 )  tmp -= u[i+N*(l_j-1)]
            ;

            l_v[i+N*l_j] = tmp
    }
```

if this is run on 1 process, it will work just like the serial code.

But if we want to run of $p > 1$ processes, then "if (l_j>0)" needs work.

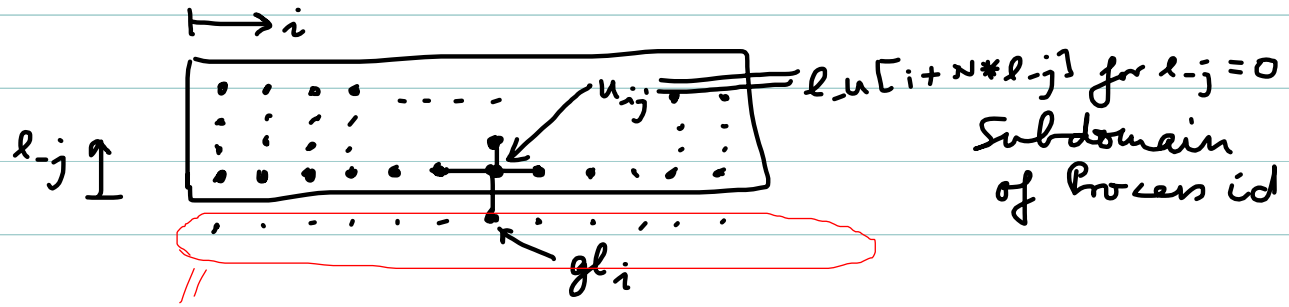
l_j == 0 is only a subdomain boundary, not necessarily on $\partial\Omega$. => For Process id > 0, we need to have an "else" case, since it is actually in interior of Ω and $u_{i,j-1}$ makes sense

If (i,j) is on Process id and we have l_j == 0, then where is $u_{i,j-1}$? It is on Process id-1. => Need to receive it from there!

Step 1 — some confusing code

```
for ( l_j = 0, ..... )
   for ( i = 0 ; ..... ) {
        tmp = 4.0 * l_u [ i + N * l_j ]
        if ( l_j > 0 )
             tmp -= l_u [ i + N * ( l_j - 1 ) ]
        else {/* l_j == 0 */
             j = l_j + id * l_N
             if ( j > 0 )
                  tmp -= gl [ i ]
             else /* on ∂Ω */
                  /* do nothing since u = 0 */
```

Notice that we need to store the data from Procesid-1 = idleft somewhere.
Terminology of "ghost points"



// vector of $gl \in \mathbb{R}^N$ on Procesid =

= one portion of length N of l_u on Procesid-1

Notice that our data structure for l_u is exactly, so that there N values are consecutive in l_u namely the last N values on Process id-1

⇐ So, each process needs to send these last N values of l_u to Process id_right = id+1

⇒ MPI_Send (&(l_u[N*(l_N-1)]), N, MPI_DOUBLE,

Code above has too complicated of an if-statement Also, we want to start preparing for doing useful work (= calculations) while waiting for communications ⇒ Want to clearly identify those portions of code that do or do not depend on any variable that needs to be received.

Step 2a - rough outline:

```
for ( l_j=1; l_j < l_N-1 ; ... )
        for (i = 0,...
                tmp -= l_u[i+N*(l_j-1)]
```

```
l_j = 0
for (i = 0; .... )
    j = l_j + id * l_N
    if (j > 0) tmp -= gl[i]
```

```
l_j = l_N-1
for [i = 0; --- )
    j = l_j + id * l_1
    if (l_i < l_N-1) tmp -= l_u[i+N*(l_j+1)]
```

Step 2 — write clean code w/ communications
= with tag

```
MPI_Recv ( gl , N, MPI_DOUBLE, idleft, 1, ...)
MPI_Recv ( gr , N, MPI_DOUBLE, idright, 2, ...)
MPI_Send ( &(l_u[N*(lN-1)]), N, DOUBLE, idright, 1
MPI_Send ( l_u , N, MPI_DOUBLE, idleft, 2, .... )
              ||
          &( l_u[0])

for ( l_j = 1 ; l_j < l_N - 1;  ...  )
    for ( i = 0; ....
        j = l_j + id * l_N
        tmp = 4.0 * l_u[i + N * l_j]
          tmp -= l_u[i + N* (l_j-1) ]
          if (i > 0) tmp -= l_u [i-1 + N*l_j]
          if (i < N) tmp -= l_u[i+1 + N * l_j]
          tmp -= l_u [i + N* (l_j+1) ]
          l_v[i+N*l_j] = tmp
}

  l_j = 0
  for (i = 0; i < N; i++) {
      j = l_j + id * l_N
      tmp = 4.0 * l_u[i+ N*l_j]
    if (j>0) tmp -= gl[i]
      : as above

l_j = l_N - 1
```