Wed. 05/09/12:

HW5:

Serial C: $N = 2048$ => time in seconds (with Intel CPU on laptop)

apparently between (134 to) 306 sec. to 384 sec.

AMD Opteron: "12 cores" ("4 cores" 660 sec.)

iter between 3192 to 3264, but $\|u - u_h\|_{L^\infty(\Omega)} \approx 7.81 \times 10^{-7}$

HW7:

It turns out that new computational hardware at UniKS will only become available next month. => no use for us!

This would have been interesting due to InfiniBand network vs. current Gigabit ethernet.

So, for HW7: Produce all C results for 4 cores as well as 12 cores!

Start early! Since 12 cores appears busy right now =>

Idea: Do on 4 cores, so you have all results and are able to write conclusions. Then during last week, run on 12 cores and add these results.

Realistic: 4 cores, 16 nodes => $4 \times 16 = 64$ processes

12 cores, 4 nodes => $8 \times 4 = 32$ processes

Ch. 3, see Exercises in Pacheco that ask you to try out if MPI_Send needs to be before MPI_Recv or not.

```
MPI_Recv (__, id+1, ... )
MPI_Send (..., id-1, ... )
```

This cannot work, since code will stop at MPI_Recv to wait for Send to take place.

```
MPI_Send (..., id-1, .. )
MPI_Recv (..., id+1, .. )
```

Will this work? Most likely!

Typically, systems have buffering enabled, so the Send will copy its message to a buffer and immediately return control to the code, which then executes the next command, here the Recv. This requires that the message fit into the buffer. Otherwise the code will deadlock, since MPI_Send will block.

More generally, we learn that MPI is a voluntary standard. So, MPI_Send can be blocking but does not have to be.

Ch. 13 Advanced Point-to-Point Communications

The standard has many additional commands that perform in more specific ways:

MPI_Ssend = "synchronous"
= blocking
→ really no reason to use

MPI_Isend / MPI_Irecv = "immediate"
= "immediately returns control"

You can pair any Send with any Recv!
But we may as well use both Isend and Irecv

Persistent Send/Recv: Initialize communication only once, then remain in communication continuously. For HW 6/7 with CG and Ax, you would need to initialize communication in main() (outside of CG!), then start each time you call Ax().

Buffered Send/Recv = you supply your own buffer. If you can guarantee that it is large enough, then there is no risk of deadlock.

Concretely, let's focus on 2 basic ones:

Blocking version: see code last time, but need to make safe (against deadlock)

Classic idea:

```
if (id%2 ==0) { /* even-numbered id */
  MPI_Recv
  MPI_Recv
  MPI_Send
  MPI_Send
}else { /* odd-numbered id */
  MPI_Send
  MPI_Send
  MPI_Recv
  MPI_Recv
}
```

There are exactly the same MPI commands as last time.

Non-blocking Communication: MPI_Isend / MPI_Irecv

```
MPI_Status   statuses[4]
MPI_Request  requests[4]
MPI_Irecv( gl, .., idleft, 1, MPI_COMM_WORLD, &(requests[0]) )
MPI_Irecv( gr, .., idright, 2,  -"-   , &(  -   [1]) )
MPI_Isend( &(l_u[N*(l_N-1)]), .., idright, 1, &(...  [2]) )
MPI_Isend( &(l_u[0]), ...  , idleft, 2, &(..   [3])
for( l_j = 1; l_j < p_(N-1) ; l_j++ )
    for( i = 0; ...                 ) {
        :
}

MPI_Waitall(4, requests, statuses)
l_j = 0
for( i = 0; ... ) {
    :
}
l_j = l_N-1
for( i = 0; ... ) {
    :
}
```

Recall that idleft = MPI_PROC_NULL on Process id = 0 and idright = MPI_PROC_NULL on id = np-1.
We use this here, so that we do not need any if statements around the Send/Recv.

Some notes:

Norm calculation: do not allocate any additional vectors!

$\|u - u_h\|_{L^\infty(\Omega)} = \sup_{(x,y)\in\Omega} |u(x,y) - u_h(x,y)|$

$= \max_{(i,j)} |u(x_i, y_j) - u_{ij}|$

```
l_norm = 0.0
for ( l_j = 0;....
  for (i = 0;..                    ) {
        /* numerical solution l_u[i + N*l_j]
           approximates u(x_i,y_j) = ufct (x[i], y[j])*/
        j = l_j + id * l_N
        new_norm = fabs ( l_u[i + N*l_j] - ufct(x[i],y[j]))
        if (new_norm > l_norm)  l_norm = new_norm
}
MPI_Reduce(& l_norm, & norm, 1, MPI_DOUBLE, MPI_MAX, 0,
                                                  MPI_COMM_WORLD)
#
#include <math.h>   → man page of math.h
ufct(x,y) = pow( sin(M_PI *x), 2) * ....
```

OR:

```
#define PI 3.14....
```

need double-precision ≈ 16 digits

from Matlab of "pi" with "format long"

OR:

```
#include <math.h>
double pi = 4.0 * atan(1.0)
```

to force output for debugging: fflush(stdout)