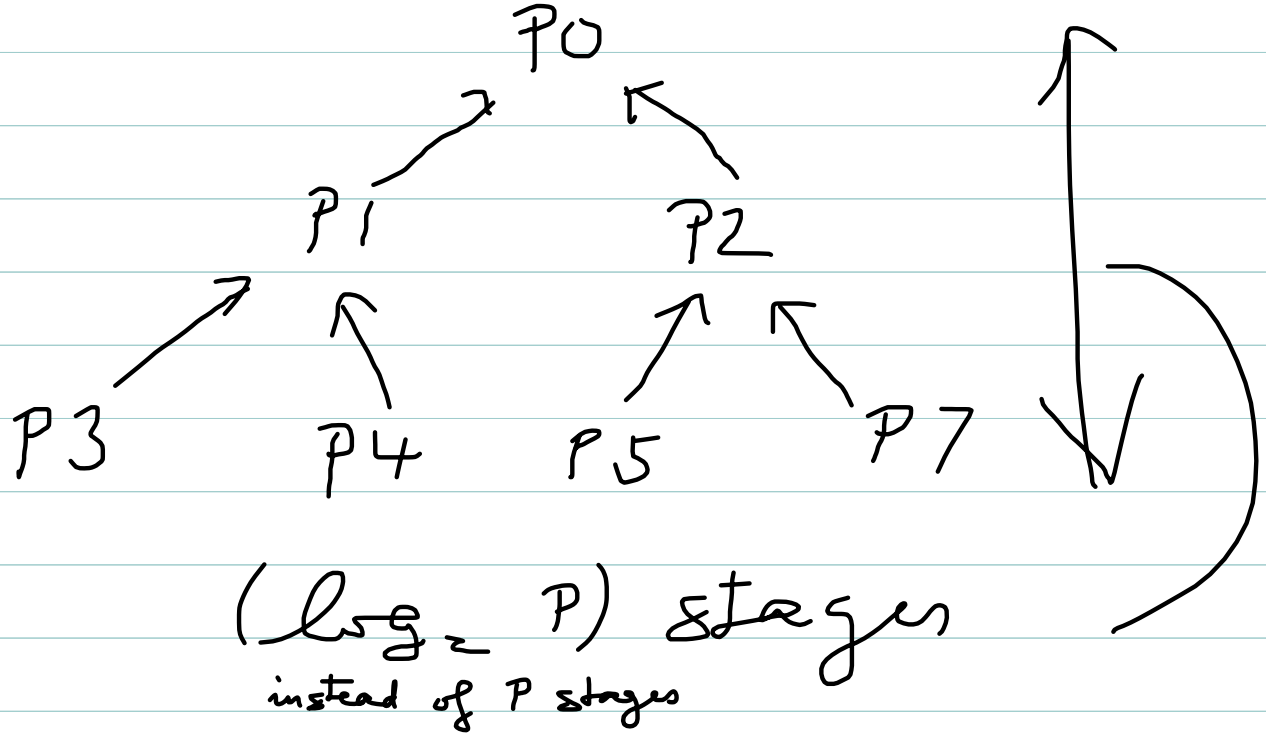
Wed., 04/18/12:

Ch. 5 Collective Communications

Sec. 5.1 Motivation:

Recall from Ch. 3 and 4 that Process 0 has to wait for communications from all other processes. During this time, the other processes are nearly always idle.

→ Idea: A communication using a tree structure should be better.



Point of MPI as a standard is to have its implementation worry about the details. For instance, different ways to organize this tree might behave differently due to whether some of the processes are on one node or not

Sec. 5.2 Broadcast

This chapter introduces commands that involve all processes (= collective) in one communicator and that accomplish important standard tasks.

Broadcast = communication from one process to all:

```
MPI_Bcast (
    void        *buffer,      <- input on root, output on all other processes
    int         count,
    MPI_Datatype datatype,    } input
    int         root,
    MPI_Comm    comm)
```

This sends data in buffer from the root process to all other processes. For collective communications, the variables need to be defined on all processes! But they may only have sensible value after receiving data.

Ex.: Assume that only one process, namely 0, has the input arguments to the method, a, b, n in trap.m4.

```
                          root
                           ↓
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD)
```

Sec. 5.3 Synchronization

Collective communication commands need to be called by <u>all</u> processes (in the communicator) that is, there should <u>not</u> be any if statement around them.

If you do this anyway, then it will have effect:

```
x=5, y=10
if ( (id%2) == 0) { /* even-numbered process */
    MPI_Bcast ( &x, 1, ... , 0,        <- root
    MPI_Bcast ( &y, 1, ... , 0,
} else { /* odd-numbered process */
    MPI_Bcast (&y, 1, ..., 0, ...
    MPI_Bcast ( &x, 1, ..., 0, ...
}
```

$\Rightarrow$ $x=5, y=10$ on all even-numbered processes

$x=10, y=5$ on all odd-numbered processes.

We notice that MPI_Bcast will execute in the order it is programmed and they behave as if synchronized.

Actual example of the deliberate application of an if-statement around MPI_Bcast is in Pacheco Sec. 7.8 Fox's algorithm on page 127

Sec. 5.4 Reduce:

Kind of an inverse of Broadcast, where all processes send data and one process ends up with the result that is of the size as the original data. => The data from all processes gets reduced in dimension by some operation.

Possible operations: MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN

Note: if input data is vector, then operations act componentwise.

```
MPI_Reduce(
  void        *operand   /* input on all processes */
  void        *result    /* output on the root */
  int         count
  MPI_Datatype datatype
  MPI_Op      operator
  int         root
  MPI_Comm    comm)
```

This is a collective command, so all processes need to call it. => output variable result needs to be defined on all processes, but in case of vector only needs to be allocated on the root process.

Ex.: trap.mk: Have local_In on all processes and want to have their sum on Process 0.

```
double local_In, In

MPI_Reduce( &local_In, &In, 1, MPI_DOUBLE
            MPI_SUM, 0, MPI_COMM_WORLD)
                     ↑
                    root
```

In has sensible value only on Process 0.

Sec. 5.6 Allreduce

Want to reduce some data on all processes by some operation and want the result to be available on all processes

$\Leftrightarrow$

MPI_Reduce ..... as above --- then:
MPI_Bcast

$\Leftrightarrow$

MPI_Allreduce
has same calling as MPI_Reduce,
but without root

Sec. 5.5 Dot Product = main example for an Allreduce and introduction to vectors in parallel programs

Mathematically, $x, y \in \mathbb{R}^n$ column vectors
On P parallel processes, split them by blocks of consecutive elements

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} l\_x \\ l\_x \\ l\_x \end{bmatrix} \begin{matrix} \} \text{ on Process 0} \\ \} \text{ on Process 1} \\ \} \text{ on Process 2} \end{matrix}$$

$n = 9$, $np = 3$, $\Rightarrow l\_n = \frac{n}{np} \Rightarrow l\_x \in \mathbb{R}^{l\_n}$

with $l\_x = \begin{bmatrix} l\_x_0 \\ l\_x_1 \\ l\_x_2 \end{bmatrix}$ on each process

Notice that we split x into consecutive blocks, which is inspired by caching, which performs best on consecutive memory

Ex.: Compute the dot product of $x \in \mathbb{R}^n$ with $y \in \mathbb{R}^n$ represented as above, and make the result available on all processes.

Recall $d = x^T y = \sum_{i=0}^{n-1} x_i y_i$

```
double serial_dot (double *x, double *y, int n){
    double d
    int i
    d = 0
    for (i=0; i<n; i++)
        d += x[i] * y[i]
    return d
}

double parallel_dot (double *l_x, double *l_y,
                     int l_n, int n )
  /* compute local dot product of l_x and l_y */
  l_d = serial_dot( l_x, l_y, l_n )
  /* then reduce by summing up */
  MPI_Allreduce ( &l_d, &d, 1, MPI_DOUBLE, MPI_SUM,
                  ... )
  return d
}
```