Wednesday 12/14/11:

HW 3:

- minor improvements and learn C:
  - command-line arguments
  - Error checking
- load-balancing when p does not does not divide n
- performance study = Part (f):
  Demonstrate that the code scales well that is, if you use p parallel processes, it should be p times as fast
  This tests both the numerical algorithm, its implementation, and the cluster being used.

Concretely, you should run the code on as many nodes as possible and try out different numbers of processes per node ⇒

So, you might want to run the code inside of directories with names like

n01ppn01, n01ppn02, n01ppn04, n01ppn08, n01ppn12,

n02ppn01, n02ppn02, etc

⇔ nXXppnYY with XX=01, 02, 04, 08, 16, 32, YY=01, 02, 04, 08, 12

So, how do know that you used enough nodes?

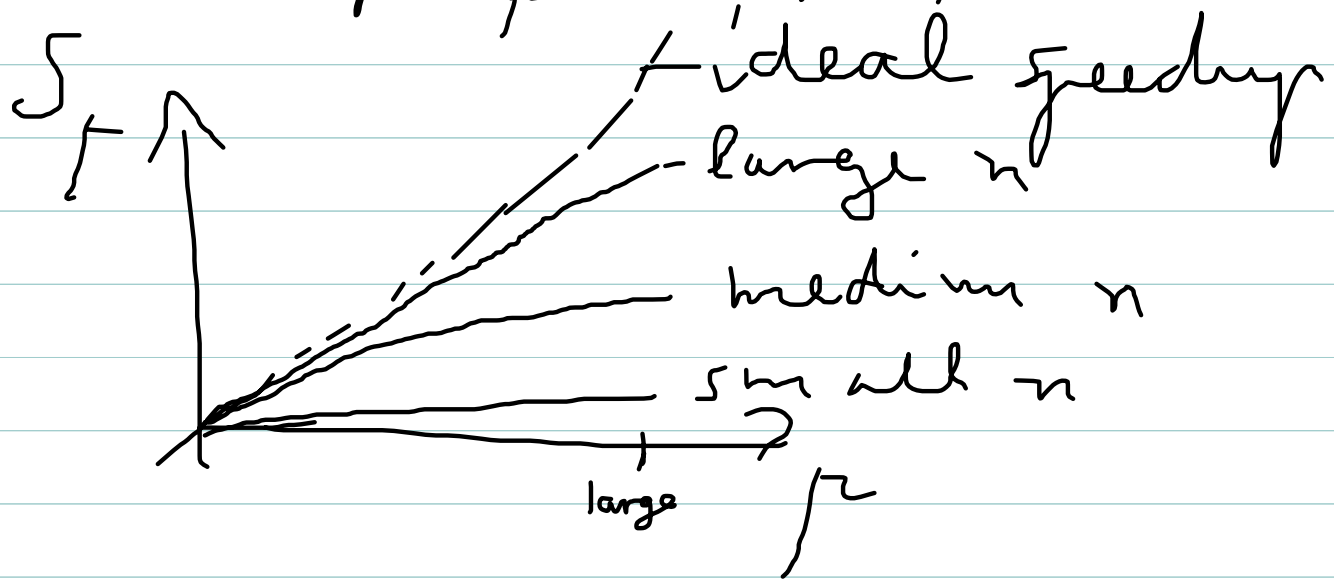
We are done if we can show good performance.
When will it perform better? $\Leftrightarrow$ When the serial job becomes larger $\Leftrightarrow$ larger n = subintervals
$\Leftrightarrow$ more work to do and more work to divide among the processes. Eventually, if very many processes are used, very little work is done on each process and simultaneously more processes need to communicate, which takes more time.

Graphical representation of speedup

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = \text{ratio}$$

of time for job of size n on 1 process vs p processes.

Ideal: $T_p = \frac{T_1}{p} \Leftrightarrow S_p = p$



See HPCF-2010-2 summary tables Table 1.1, 1.2

for the idea of XX nodes and YY processes per node.
See Section 4 for speedup (and efficiency) plots.

Always: Result must be correct numerically!
$\Leftrightarrow$ error $= I - I_n \approx O(h^2)$

C issue of command-line inputs:

Want to be able to pass a, b, n from the command-line like

```
./trap 0.0 1.0 1024
```

example for next page

or in submission script

```
myjob=./trap 0.0 1.0 1024
```

```
int main (int argc, char *argv[])
```
$\Leftrightarrow$ ----- `char **argv )`

The int argc gives the count of input arguments.
the char *argv[] gives argc many strings,
each of which is actually an array of characters.

Test output:

```
printf("%d\n", argc);
for (i=0; i<argc; i++) {

  printf("%s\n", argv[i])

}
```

will print also "./trap" as the "0th" argument

4 ← argc

./trap

0.0

1.0

1024

Have definitions like

int n

float a, b

a = atof (arg[1]);

b = atof (argv[2])

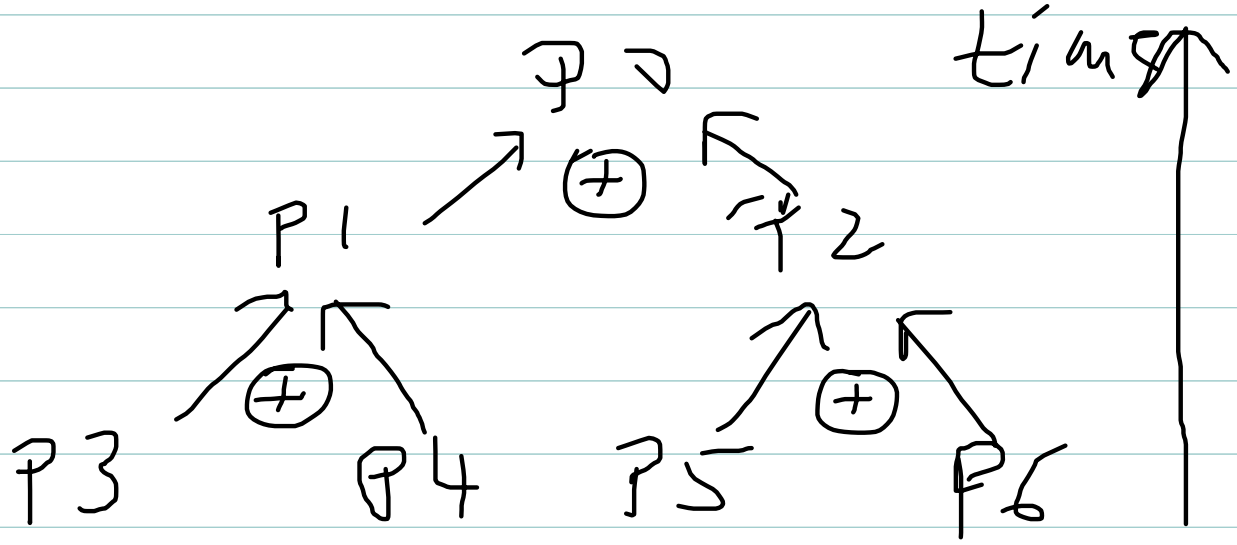
n = atoi (argv[3])

Ch. 5 Collective Communications

Sec. 5.1 Motivation:

- a for loop like in the trapezoidal program of Ch. 4 !
- most processes are idle most of the time → inefficient

Idea: tree-structured communication



Idea of using MPI for portable code is to have MPI choose the exact structure at run time, depending on which node has each process.

Sec. 5.2 Broadcast:

Idea in context of trap. example
Only P0 has value of n and needs to send it to all processes ⇒

```
MPI_Bcast (
    void * buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm )
```

here

```
MPI_Bcast (&n, 1, MPI_INT, 0, ...)
```

(↑ address of n)

Notice: buffer is input on root process, and output on all others.

Sec. 5.4 Reduce:

The tree-structured example is a Reduce operation, since a piece of information (= 1 on each process = local_In) needs to become 1 piece of information on Process 0 (= In), so there must be some 'loss of information'

Summing all local_In to find In is one example.

Others could be to find the product, the maximum, or the minimum.

MPI_Reduce(

```
void        *operand    /* input */
void        *result     /* output */
int         count,
MPI_Datatype datatype,
MPI_Op      operator,
int         root,
MPI_Comm    comm )
```

Ex.: Sequential code:

```
MPI_Reduce( &integral, &total,
    1, MPI_FLOAT, MPI_SUM,
    0, MPI_COMM_WORLD )
```

Table 5.4 in Pacheco: MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN

Notice: MPI_Reduce is collective (⇒ needs to be called on all processes. ⇒ total needs to be defined on all processes, but it will not have a sensible value on any process except 0.

Sec. 5.6 Allreduce:

How about if you want/need the value of the reduced quantity on all processes?

Could do Reduce to P0, then Bcast.

$\Leftrightarrow$

MPI_Allreduce

has same calling sequence as MPI_Reduce, except without root.