Tuesday 04/24/12:

HW plan:

Before Wed. download files and put in proper place

tar -zxf ver0.0 suppliedfiles.tgz

zip/unzip — "expand"

Have a look at the files before lecture and ask questions

Test "make".

Fri: Have working code and show to Stefan!

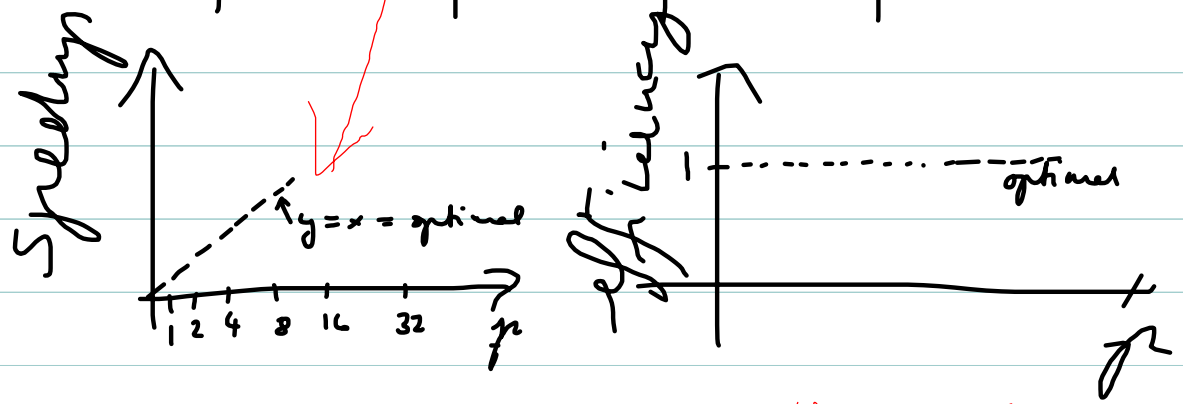
→ Sit together as team!

Mo.: Have all results and assembled as table

Wed. morning: Submit HW

time in seconds:

	1 node	2 nodes	4 nodes
1 ppn	16	8	6
2 ppn	9	4	2
4 ppn	5	3	1
8 ppn	2	1	0.5



Use as many processes per node as possible, as the red numbers above

or as in Figure 4.5 of tech. rep.

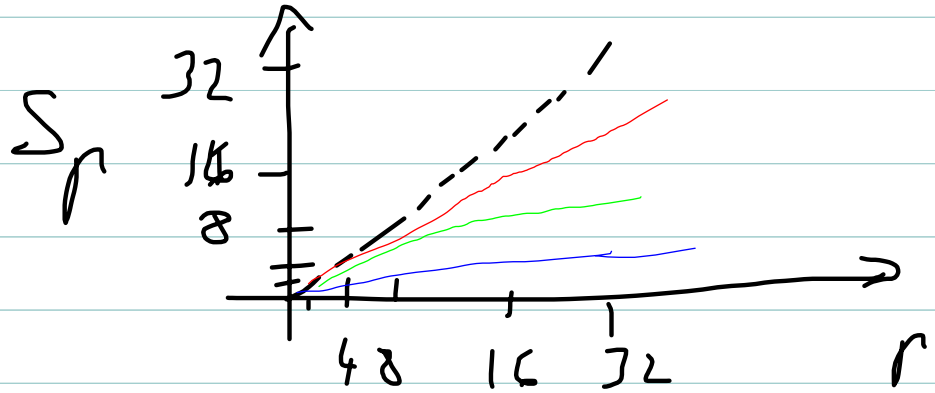
Def.: $S_p = \frac{T_1}{T_p}$, $E_p = \frac{S_p}{p}$

What does the best possible speedup plot look like?

basic answer: "optimal S_p" is linear

more subtle answer:

- We want several cases of problem size to get a feel for performance
- Specifically, we want one bad case, one excellent case, and one or two in between



Use Matlab script for the plot!

Idea is to reuse this script in future HW by just changing the numbers!

Currently, the public partition has a mixture of 4 core nodes (2 dual-core CPUs) and 12 core nodes (2 six-core CPUs).

Any mixture in one run would behave strangely.

Also, any mixture from one run to the next does not allow to compare performance, such as speedup.

⇒ Use either --constraint = 4 cores or
= 12 cores for all studies

Ch. 5 Collective Communications

Last time MPI_Bcast, MPI_Reduce, MPI_Allred.

Sec. 5.7 Gather and Scatter:

Example with vectors last time

$x \in \mathbb{R}^n$, $l\_x \in \mathbb{R}^{l\_n}$ on each process

$l\_n = n/np$

Want to assemble x on Process 0 (for test output or some other purpose)

```
MPI_Gather (
    void         *send_data
    int          send_count
    MPI_Datatype send_type
    void         *recv_data
    int          recv_count
    MPI_Datatype recv_type
    int          root
    MPI_Comm     comm )
```

```
MPI_Gather ( l_x, l_n, MPI_DOUBLE,
             x, l_n, MPI_DOUBLE,
             0, MPI_COMM_WORLD )
```

Notice: recv_count is the number of elements from each process $\Rightarrow$ l_n, not n

Since MPI_Gather is collective, it needs to be executed on all processes ⇒ x needs to be defined on all processes. But it needs to have memory allocated only on Process 0

```
double **x
if (id == 0)
    x = malloc(n * sizeof(double))
```

other options:

```
x = calloc(n, sizeof(double))
```

yet another option:

```
double x[n]
```

would work if n is constant or under some conditions. Clearly, one problem would be that x would be defined and allocated on all processes

MPI_Scatter is the inverse operation of MPI_Gather

```
MPI_Scatter( x, l_n, MPI_DOUBLE,
             l_x, l_n, MPI_DOUBLE,
             0, MPI_COMM_WORLD)
```

MPI_Allgather does exactly the same as MPI_Gather, but it makes the result available on all processes

Same syntax, except no 'root' argument.

→ See man pages! Look for how it defines the send_count, recv_count.

Do and contrast to Pacheco examples:

```
MPI_Gather( l_A, n*l_n, MPI_DOUBLE,
            A, n*l_n, MPI_DOUBLE,
            0, MPI_COMM_WORLD )
```

matrix $A \in \mathbb{R}^{n \times n}$ (square), assume np divides $n \Rightarrow l\_n = n/np$ is integer

$$A = \begin{bmatrix} \underline{l\_A} \\ \underline{l\_A} \\ \underline{l\_A} \\ l\_A \end{bmatrix} \text{ with } l\_A \in \mathbb{R}^{l\_n \times n}$$

or

$$A = \left[\, l\_A \,\middle|\, l\_A \,\middle|\, l\_A \,\middle|\, l\_A \,\right] \text{ with } l\_A \in \mathbb{R}^{n \times l\_n}$$

Pacheco shows rowwise division $A = \begin{bmatrix} = \\ = \end{bmatrix}$

since C does it like that

double A[n][n], l_A[l_n][n]

dynamic memory allocation for matrix

```
double **l_A
l_A = malloc(l_n * sizeof(double *))
for(i=0; i<l_n; i++) {
    l_A[i] = malloc(n * sizeof(double))
}
```

⇒ l_A ij mathematically is l_A[i][j]

In this sense, this is row-oriented

Only each row is consecutive in memory, not l_A overall ⇒ this means that the above example for MPI_Gather is wrong; it would only be applicable for double l_A[l_n][n]

But we want to use dynamic memory allocation and MPI_Gather

⇒ Need to use 1-D arrays! storage

Then, we may as well do column-oriented