Lecture 6: Introduction to MPI programming

MPI (message passing interface)

MPI is a library standard for programming distributed memory

- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality
- Collaborative computing by a group of individual processes
- Each process has its own local memory
- Explicit message passing enables information exchange and collaboration between processes

More info: `http://www-unix.mcs.anl.gov/mpi/`

MPI basics

- The MPI specification is a combination of MPI-1 and MPI-2
- MPI-1 defines a collection of 120+ commands
- MPI-2 is an extension of MPI-1 to handle "difficult" issues
- MPI has language bindings for F77, C and C++
- There also exist, e.g., several MPI modules in Python (more user-friendly)
- Knowledge of entire MPI is not necessary

MPI language bindings

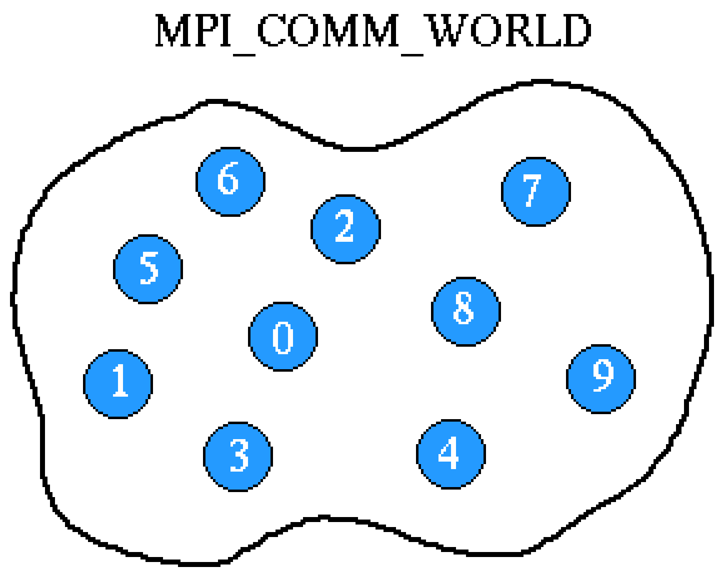
C binding

```
#include <mpi.h>

rc = MPI_Xxxxx(parameter, ... )
```

Fortran binding

```
include 'mpif.h'

CALL MPI_XXXXX(parameter,..., ierr)
```

MPI communicator



- An MPI communicator: a "communication universe" for a group of processes
- `MPI_COMM_WORLD` – name of the default MPI communicator, i.e., the collection of all processes
- Each process in a communicator is identified by its rank
- Almost every MPI command needs to provide a communicator as input argument

MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between `0` and `#procs-1`
- The rank value is used to distinguish one process from another
- Commands `MPI_Comm_size` & `MPI_Comm_rank` are very useful
- Example

```
int size, my_rank;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

if (my_rank==0) {
  ...
}
```

The 6 most important MPI commands

- `MPI_Init` - initiate an MPI computation
- `MPI_Finalize` - terminate the MPI computation and clean up
- `MPI_Comm_size` - how many processes participate in a given MPI communicator?
- `MPI_Comm_rank` - which one am I? (A number between `0` and `size-1`.)
- `MPI_Send` - send a message to a particular process within an MPI communicator
- `MPI_Recv` - receive a message from a particular process within an MPI communicator

MPI "Hello-world" example

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I’ve rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

MPI "Hello-world" example (cont'd)

- Compilation example: `mpicc hello.c`
- Parallel execution example: `mpirun -np 4 a.out`
- Order of output from the processes is not determined, may vary from execution to execution

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```

The mental picture of parallel execution

The same MPI program is executed concurrently on each process

Process 0   Process 1 $\cdots$ Process P-1

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Synchronization

- Many parallel algorithms require that none process proceeds before all the processes have reached the same state at certain points of a program.
- Explicit synchronization

  ```
  int MPI_Barrier (MPI_Comm comm)
  ```

- Implicit synchronization through use of e.g. pairs of `MPI_Send` and `MPI_Recv`.
- Ask yourself the following question: "*If Process 1 progresses 100 times faster than Process 2, will the final result still be correct?*"

Example: ordered output

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I've rank %d out of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

Example: ordered output (cont'd)

Process 0

```c
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I've rank %d out of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

Process 1

```c
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I've rank %d out of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

. . .

Process P-1

```c
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I've rank %d out of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

- The processes synchronize between themselves P times.
- Parallel execution result:

```
Hello world, I've rank 0 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
```

MPI point-to-point communication

- Participation of two different processes
- Several different types of send and receive commands
  - Blocking/non-blocking send
  - Blocking/non-blocking receive
  - Four modes of send operations
  - Combined send/receive

The simplest MPI send command

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm);
```

This blocking send function returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

The simplest MPI receive command

```
int MPI_Recv(void *buf, int count
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status);
```

- This blocking receive function waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or `MPI_ANY_SOURCE`), message tag (or `MPI_ANY_TAG`).
- Receiving fewer `datatype` elements than `count` is ok, but receiving more is an error.

MPI message

An MPI message is an array of data elements "inside an envelope"

- *Data*: start address of the message buffer, counter of elements in the buffer, data type
- *Envelope*: source/destination process, message tag, communicator

MPI_Status

The source or tag of a received message may not be known if wildcard values were used in the receive function. In C, `MPI_Status` is a structure that contains further information. It can be queried as follows:

```
status.MPI_SOURCE

status.MPI_TAG

MPI_Get_count (MPI_Status *status,
               MPI_Datatype datatype,
               int *count);
```

Example of `MPI_send` and `MPI_recv`

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

Example of MPI_send/MPI_recv (cont´d)

Process 0

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

Process 1


```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```


$\cdots$

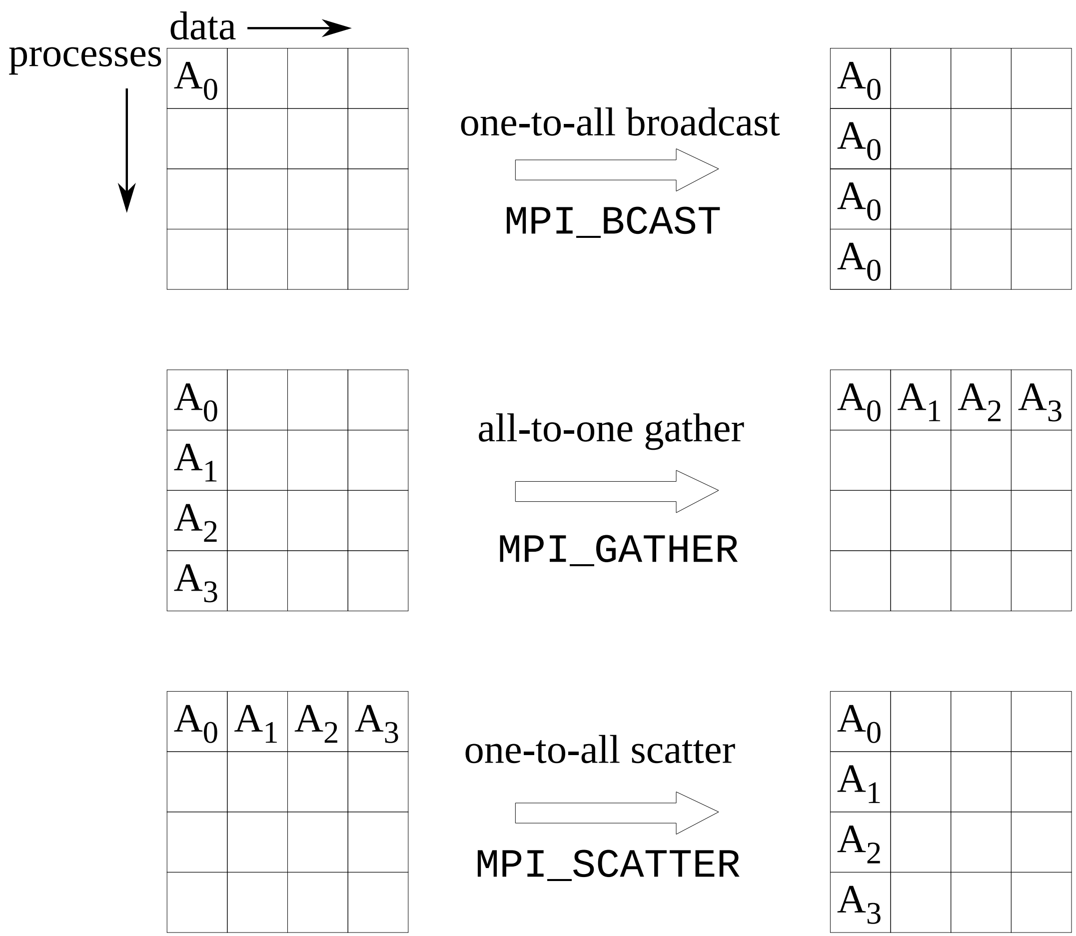
Process P-1


```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

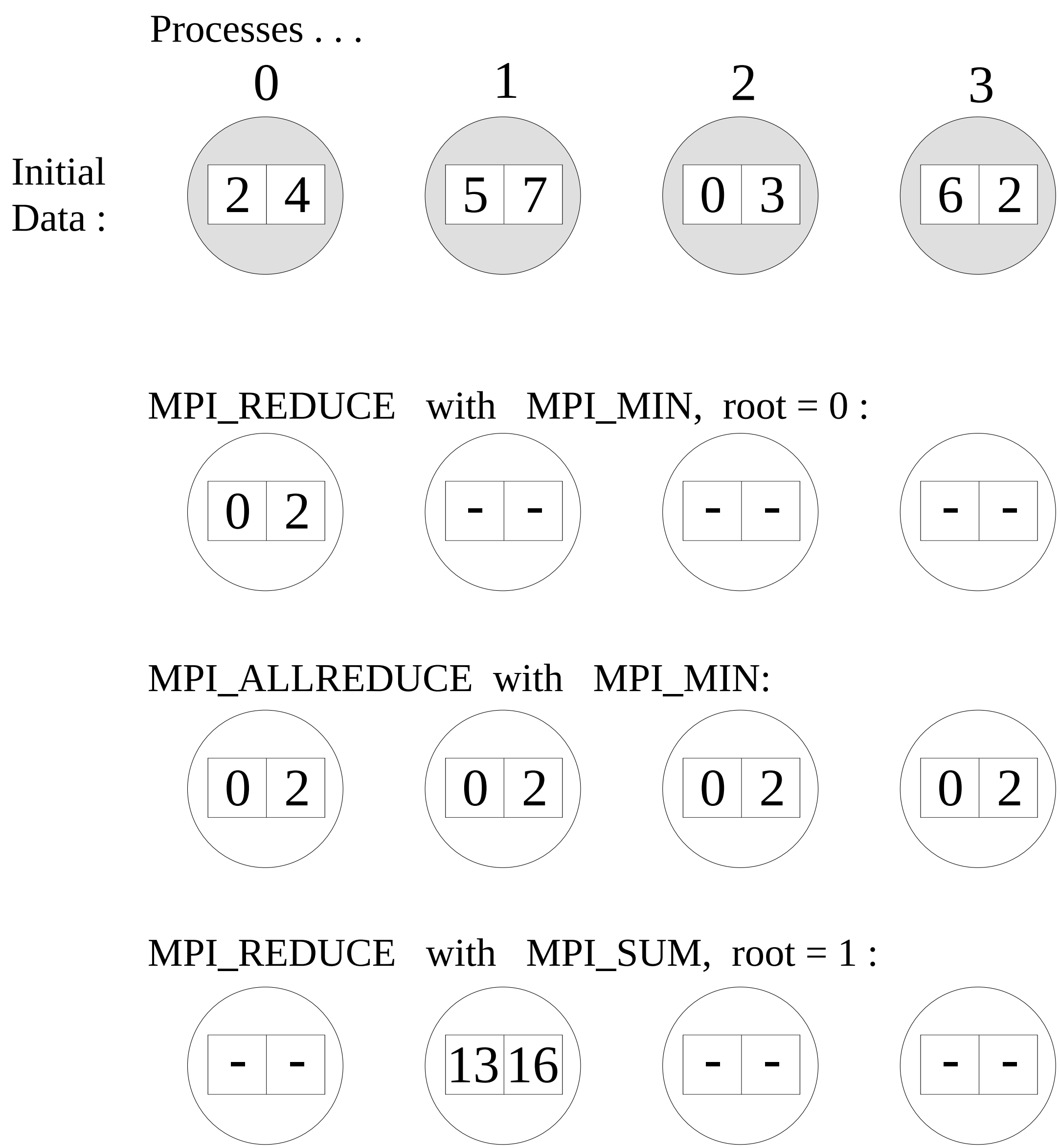

- Enforcement of ordered output by passing around a "semaphore", using `MPI_send` and `MPI_recv`
- Successful message passover requires a matching pair of `MPI_send` and `MPI_recv`

MPI collective communication

A collective operation involves *all* the processes in a communicator: (1) synchronization (2) data movement (3) collective computation



Collective communication (cont´d)



Computing inner-product in parallel

- Let us write an MPI program that calculates inner-product between two vectors $\vec{u} = (u_1, u_2, \ldots, u_M)$ and $\vec{v} = (v_1, v_2, \ldots, v_M)$,

$$c := \sum_{i=1}^{M} u_i v_i$$

- Partition $\vec{u}$ and $\vec{v}$ into P segments each of sub-length

$$m = \frac{M}{P}$$

- Each process first concurrently computes its local result
- Then the global result can be computed

Making use of collective communication

```
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

my_start = M/num_procs*my_rank;
my_stop = M/num_procs*(my_rank+1);

my_c = 0.;
for (i=my_start; i<my_stop; i++)
  my_c = my_c + (u[i] * v[i]);

MPI_Allreduce (&my_c, &c, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
```

Reminder: steps of parallel programming

- Decide a "breakup" of the global problem
  - functional decomposition – a set of concurrent tasks
  - data parallelism – sub-arrays, sub-loops, sub-domains
- Choose a parallel algorithm (e.g. based on modifying a serial algorithm)
- Design local data structure, if needed
- Standard serial programming plus insertion of MPI calls

Calculation of π

Want to numerically approximate the value of π

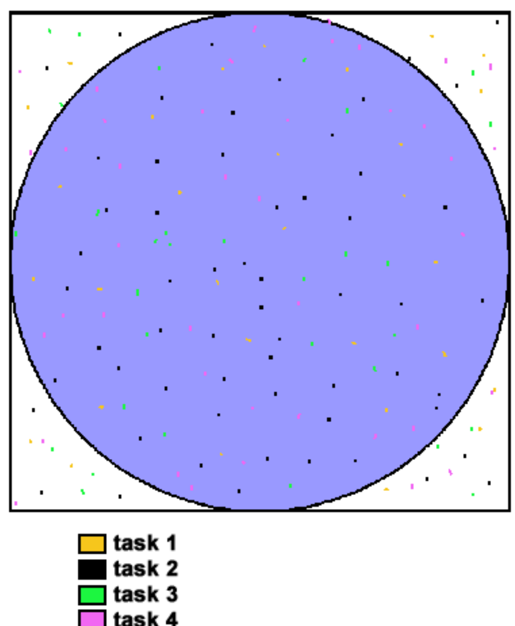
- Area of a circle: $A = \pi R^2$
- Area of the largest circle that fits into the unit square: $\frac{\pi}{4}$, because $R = \frac{1}{2}$
- Estimate of the area of the circle $\Rightarrow$ estimate of π
- How?
  - Throw a number of random points into the unit square
  - Count the percentage of points that lie in the circle by

$$\left((x - \frac{1}{2})^2 + (y - \frac{1}{2})^2\right) \leq \frac{1}{4}$$

  - The percentage is an estimate of the area of the circle
- $\pi \approx 4\,A$

Parallel calculation of π

```
num = npoints/P;
my_circle_pts = 0;

for (j=1; j<=num; j++) {
  generate random 0<=x,y<=1
  if (x,y) inside circle
    my_circle_pts += 1
}

MPI_Allreduce(&my_circle_pts,&total_count,
              1,MPI_INT,MPI_SUM,
              MPI_COMM_WORLD);

pi = 4.0*total_count/npoints;
```



The issue of load balancing

What if `npoints` is not divisible by `P`?

- Simple solution of load balancing

```
num = npoints/P;
if (my_rank < (npoints%P))
  num += 1;
```

- Load balancing is very important for performance
- Homogeneous processes should have as even disbribution of work load as possible
- (Dynamic) load balancing is nontrivial for real-world parallel computation

Exercises

- Write a new "Hello World" program, where all the processes first generate a text message using `sprintf` and then send it to Process 0 (you may use `strlen(message)+1` to find out the length of the message). Afterwards, Process 0 is responsible for writing out all the messages on the standard output.
- Write three simple parallel programs for adding up a number of random numbers. Each process should first generate and sum up locally an assigned number of random numbers. To find the total sum among all the processes, there are three options:
  - Option 1: let one process be the "master" and let each process use `MPI_Send` to send its local sum to the master.
  - Option 2: let one process be the "master" such that it collects from all the other processes by the `MPI_Gather` command.
  - Option 3: let one process be the "master" and make use of the `MPI_Reduce` command.
- Exercise 4.8 of the textbook.
- Exercise 4.11 of the textbook.