Intro to Parallel Computing

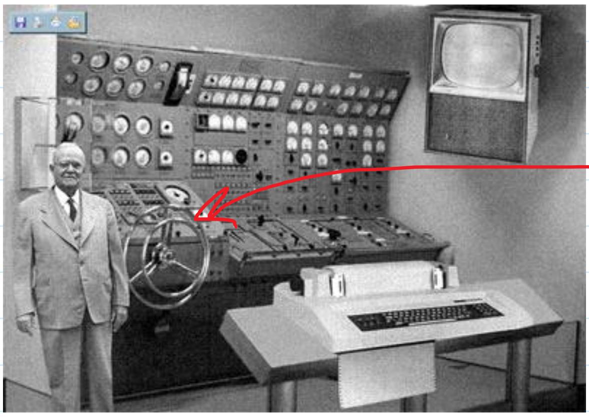
January 14, 2014 7:02 PM

Announcements

- course outline is on OWL, make sure you have access
- we will use C/C++ and MPI in this course so you need access to a computer running these
- a Linux-like environment is needed
- WSL (Windows Subsystem for Linux) should work for program development if you install the MPI runtime & developer packages in setup.
- on Macs, you can probably get by using the XCode package. There is also a MPI implementation you will need
- there are some tips on OWL in the "SetUp" file
- you need to get a SharcNet account
  → see SharcNet website
  (a line terminal is needed to access SharcNet computers)

Lecture 1

Back in the day computers looked like → (not really...)



So you could play "Mario Chariot"

	1980	1995	2005	2015
	commodore vic 20	gateway pentium PC	Dell pentium4 PC	lenovo i7 PC
CPU	1 MHz	100 MHz	2 GHz	4 X 3 GHz
Mem	5 kByte RAM 20 kByte ROM	4 Mbyte	1 GByte	4 GByte
				GPU: 1526 x 1GHz (nvidia geforce 770)

×100 ×200 ×1.5

This is the bigger CPU gain i.e. multi core processor

This is by far the biggest improvement

Reasons to parallize
→ time (clock time)
→ memory

Classification of parallelization according to data & instruction schemes

	same data	multiple data
single instruction	SISD (serial)	SIMD
multiple instructions	MISD	MIMD

note: different computer architecture may be required for different schemes but some architectures can run different schemes

eg ① SISD

Compute $F_1 = \sum_n f_n$

Algorithm

$$F_1 = f_1$$
$$F_1 \mathrel{+}= f_2$$
$$F_1 \mathrel{+}= f_3$$
$$\vdots$$
$$F_1 \mathrel{+}= f_n$$

time

eg ② SIMD

Compute $A = 4 \cdot B$ where A, B $1 \times n$ arrays
n divisible by 3

Algorithm

CPU1	CPU2	CPU3
A[1] = 4 * B[1]	A[2] = 4 * B[2]	A[3] = 4 * B[3]
A[4] = 4 * B[4]	A[5] = 4 * B[5]	A[6] = 4 * B[6]
⋮	⋮	⋮
A[n-2] = 4 * B[n-2]	A[n-1] = 4 * B[n-1]	A[n] = 4 * B[n]

time

The division of data is clearly not unique. Different divisions may result in faster / slower code.

eg ③ MISD

Compute $F_\mu = \frac{1}{N} \sum_n^N f_n^\mu$ (moments problem)

CPU1	CPU2	…	CPU N
$F_1 = f_1$	$F_2 = f_1^2$	…	$F_N = f_1^N$
$F_1 \mathrel{+}= f_2$	$F_2 \mathrel{+}= f_2^2$		$F_N \mathrel{+}= f_2^N$
$F_1 \mathrel{+}= f_3$	⋮		⋮
⋮			

time

time ↓

$F_1 \mathrel{+}= f_3$ $\qquad \vdots \qquad \circ$

$\vdots$

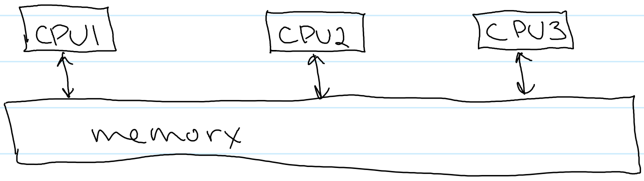
$F_1 \mathrel{+}= f_n \qquad F_2 \mathrel{+}= f_n^2 \qquad F_N \mathrel{+}= f_n^N$

eg ④ MIMD different CPUs have different tasks & act on different data sets

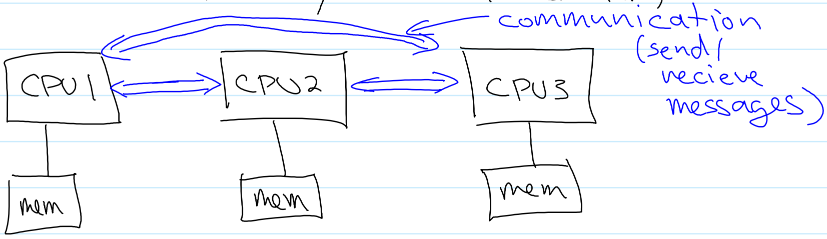
CPU 1-10	CPU 11-20
Solve Diffusion Equation and Write Data at each time step to a file	Analyze output files to calculate toxicity at a given location

Classification of parallel computing according to memory

Shared Memory (typical desktop today)



Distributed Memory (beowulf cluster)



In practise, most systems today are some combination. However, it is useful to develop algorithms based on a single paradigm in order to make portable code.

MPI is based on a distributed memory paradigm. It provides a library of functions to facilitate communication between processors.

$$\text{MPI} = \begin{cases} \text{Message} \\ \text{Passing} \\ \text{Interface} \end{cases}$$

MPI

- library containing ≈ 100 functions
- ~6 are really important :
  1) initialize MPI
  2) find # of processes (specified when program submitted)
  3) find out which process I am
  4) send a message
  5) recieve a message
  6) terminate MPI (essential. Make sure you always do this! MPI is C based not C++)

main point ⇒

⇒

Eg. "Hello World" program in C (source code is on OWL)

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{
   int noprocs, nid;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &nid);

   printf("Hello from processor %i of %i\n",nid,noprocs);

   MPI_Finalize();
}
```

MPI header files (always need this)

starts MPI and code communications on all processors

runs on each node indep-endently

close of all the processes

comments :

MPI_Init must be called (exactly) once before all other MPI commands

MPI_Finalize should be called after all other MPI commands

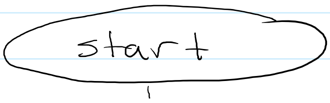
MPI_Comm_size finds out number of processors (put into noprocs)

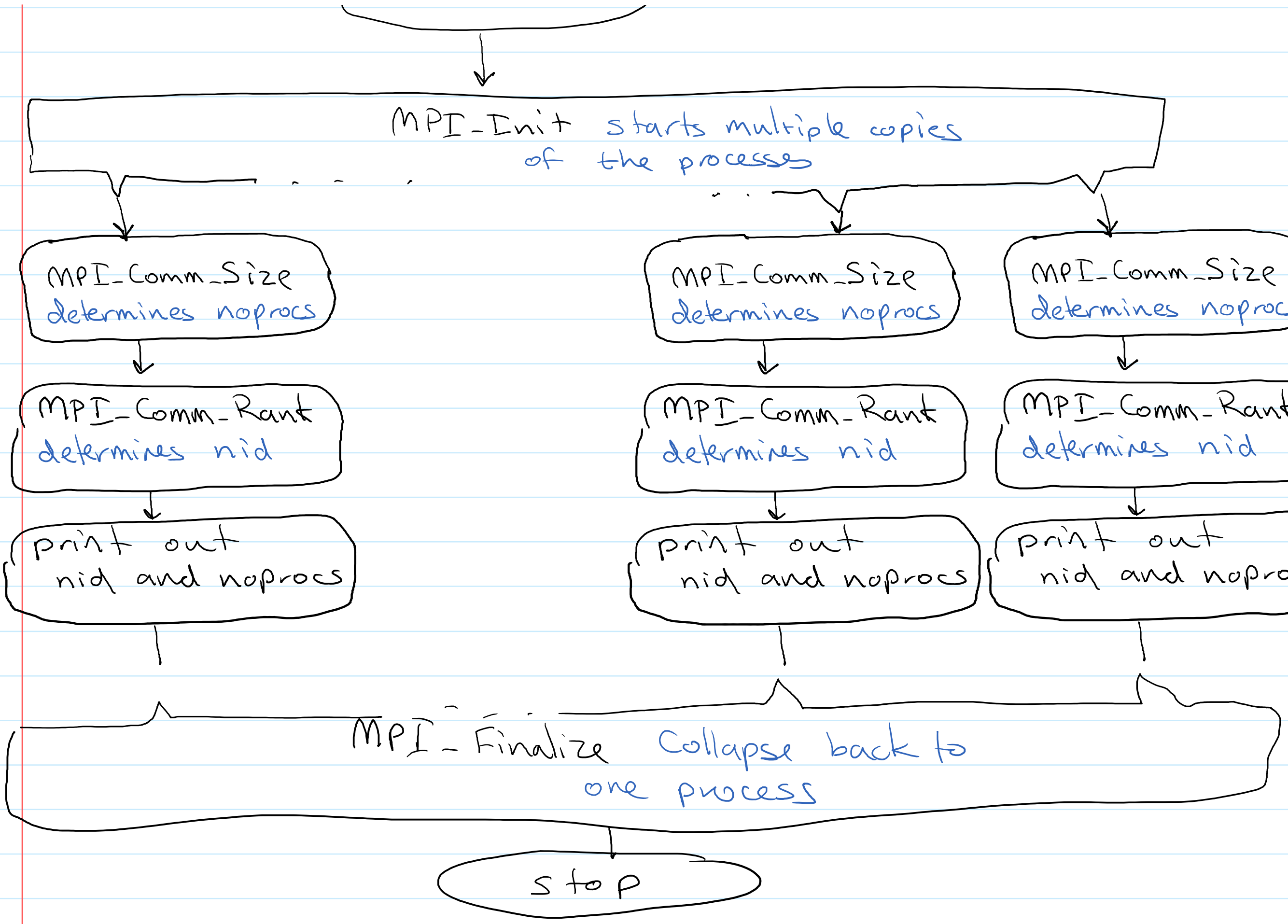
MPI_Comm_rank finds out which process I am

nid∈{0,..., nprocs −1}

MPI_COMM_WORLD is a MPI "communicator"

A flow-chart of what happens:




MPI_Init starts multiple copies of the processes
MPI_Comm_Size determines noprocs
MPI_Comm_Size determines noprocs
MPI_Comm_Size determines noprocs
MPI_Comm_Rank determines nid
MPI_Comm_Rank determines nid
MPI_Comm_Rank determines nid
print out nid and noprocs
print out nid and noprocs
print out nid and noprocs
MPI_Finalize Collapse back to one process
stop

Note that once multiprocesss are running they are independent and variables like "nid" are different for each process.

printf statement is executed on each processor independently (as are previous 2 commands) and there is a different local version of nid and noprocs on each processor, potentially with different values

The different processes are not typically in sync so the output may come in any order and be different if you run the code again.

Typical compile line is something like

```
mpicc -o hello.o hello.c
```

Typical run command is

specify # of processes

```
mpirun -np 4 ./hello.o
```

Expected Output:

```
Hello from processor 0 of 4
Hello from processor 1 of 4
Hello from processor 2 of 4
Hello from processor 3 of 4
```

} perhaps not in this order

MPI is a library written in C. It also has a Fortran version interface. There was a long-term effort to get a C++ version but it was buggy, largely shunned in favor of the C version and was abandoned as unnecessary (C++ can just use the C library).

For C++ it is always a good idea to try to follow the RAII paradigm and try to automate cleanup using class destructors to take care of it. To make our C and C++ code as similar as possible we will first rewrite hello.c to put MPI variables into a data structure and add a startup function:

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

struct mpi_vars {  ← data structure type declaration
  int NProcs;
  int MyID;
};

struct mpi_vars mpi_start(int argc, char** argv)
{
  struct mpi_vars this_mpi;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &this_mpi.NProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &this_mpi.MyID);

  return this_mpi;
}

int main(int argc, char** argv)
{
   struct mpi_vars this_mpi = mpi_start(argc, argv);

   printf("Hello from processor %i of %i\n",
          this_mpi.MyID,this_mpi.NProcs);
```

↙ initialize MPI data structure using start function

```
    MPI_Finalize();

    return 0;
 }
```

Note: 1. Only one instance of the mpi-vars should be created (on each MPI thread)

2. The mpi-start function should only ever be called once.

3. In C, MPI_Finalize() should always be called before you exit the program

Translation of this to C++ is then very straightforward:

```
#include <iostream>
#include "mpi.h"

class MPI_stuff
{
public:
  int NProcs;     } our mpi info
  int MyID;

  MPI_stuff(int &argc, char** &argv)  ← put MPI initialization
  {                                      into class constructor
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyID);
  }

  ~MPI_stuff()          ← put MPI cleanup
  {                       into class destructor
    MPI_Finalize();
  }
};


int main(int argc, char** argv)
{
  MPI_stuff the_mpi(argc, argv);  ← MPI will be started automatically
                                      by the MPI_stuff constructor

  std::cout << "Hello from processor " << the_mpi.MyID << " of "
       << the_mpi.NProcs << std::endl;

   return 0;← the_mpi variable goes out of scope here
                              will be called to close off MPI
    return 0;
}
```

← the_mpi variable goes out of scope here so the destructor will be called to close off MPI

Note: 1) The class MPI_stuff should only ever have <u>one</u> instance in existence (but this one instance may/will have different values for MyID on each processor)

2) MPI_Finalize will automatically be called when the_mpi variable goes out of scope (useful if program exits in unexpected manner too).

3) Compilation is typically something like

mpiCC -o hello.o hello.cpp

note capital C's here

"Homework"

1. Sign up for a SharcNet account and online "training seminar"
2. Try to get Hello world running on your machine