780.70 Wednesday Class

* Last period of the quarter: Finish with some parallel processing and (maybe) a final look at Qt.
  * Go through the MPI examples (3 of them).
  * Then you can work on how to convert a code from last quarter or this to run with MPI or else do the Qt exercise.

The Qt exercise involves code for a simple display application for random walks plus a finished, ‘upgraded’ version. Your job is to enhance the simple version to add the features of the fancy version (or different features at your own discretion).

* Brief overview of MPI based on Robert Bell’s Ames lecture notes and the 675 class lectures (see links on Session 79 page or 780 page).

* How do we solve a computational physics problem faster?
  * Some solutions:
    1. use a faster computer
    2. change to better theories and methods to solve the problem
    3. implement faster numerical algorithms
    4. use a smarter compiler (i.e., one that optimizes better).

Here: Introduce parallelism
  * can be applied at many levels
  * will consider one of the simplest: m processes running on m different computers (or processors) can, in principle, run a computation m times as fast.

We’ll consider Multiple Instruction Multiple Data (MIMD)
  * collection of autonomous processors, each able to run a program self-paced
  * asynchronous, but can be programmed to behave "synchronously,"
How do we carry out such a program? Here are two ways to run the same code on multiple machines, dividing up the computational load:

1. if computer X then
   call subroutine 1
else if computer Y then
   call subroutine 2

2. for (int i = computer_id; i < N; i += # of computers)
   do stuff to calculate...

- If there are m computers with ids from 0 to M = # of computers
  and there are N steps to do, we loop through the
  steps starting at the id number and skipping every m steps.

- Message passing Interface or MPI
  - version 1 (MPIL1.1) developed 1993-94 by 280 people at 10 organizations
  - version 2 (MPIL2.0) developed 1995-97

A "message" is data passed between processes in a distributed
memory environment, that is, each processor has local
memory, which cannot be accessed by other processors.

MPI goals: source code portability, efficient implementation, support for heterogeneous parallel architectures, ...

MPI naming conventions vary between Fortran, C, and C++, which may cause some issues when consulting documentation (which tends to give the Fortran and/or C versions).
The basic form of the commands will be:

- `mpi``::``Class``::``Action``::``subset`` (associated with a class)
- `mpi``::``Action``::``subset``

Example: the standard `mpi``::``send``

```c
mpil_send(-)
```
```fortran
mpi_sendv(-)
```
```cpp
mpi::Comm::Send(-)
```

The programmer sees a group of processes allowed to communicate with each other:
- `mpi``::``Comm``::``world`` is the "communication" for all of your processes (as opposed to dividing them into groups).
- Each process has a unique id number from 0 to `num_procs-1``.

Point-to-Point Communications
- Send messages one-to-one between processes
- These can be "blocking" or "non-blocking" depending on whether
  the return from a call implies the operation has completed or if the
  return is immediate (and user must test for completion).

Fields in a message:
- **source** - the process sending the message (implicitly determined)
- **destination** - the receiving process, explicitly specified by sending process
- **tag** - an id used by the receiving process to distinguish between messages
- **communicator** - a logical group of available processes; each is
  given a unique integer ("rank") in the group
- **data** - the data of the calculation (may be an integer, many integers,
  many doubles, ... a user-defined structure)
  Along with data comes a count and a datatype
6/5/04

data types include:
  MPI::CHAR char
  MPI::INT signed int
  MPI::DOUBLE double
  and so on

See the example codes for the specific form of send and receive messages. (mpi_send_receive)

Skeleton of a program

```
#include <mpi.h>

int main()
{
  MPI::Init( int argc , char *argv )
  
  do stuff
  
  MPI::Finalize()
}
```

Useful after Init: (assume COMM_WORLD here)
  int MPI::COMM_WORLD::Get_size()
  int MPI::COMM_WORLD::Get_rank()

See examples for how processes coordinate.

Collective Communications
  called by all processes in a communicator
  broadcast, scatter, gather
  "barrier" holds all processes until all have "arrived"
  See mpi_calculate_pi example.