ACM/CS 114
Parallel algorithms for scientific applications

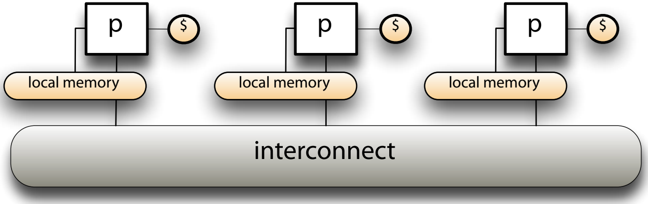
Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Distributed memory parallelism

- recall the generic layout of a distributed memory machine



  - each processor has its own private memory space
  - processors communicate via the interconnect substrate
- the programming model
  - program consists of a collection of *p* named processes
  - each process has its own instruction stream and address space
  - logically shared data must be partitioned among the processors
  - communication and synchronization must be orchestrated explicitly
  - processes communicate via explicit data exchanges

`MPI` – the survivor

- the *de facto* standard for writing parallel programs using message passing
  - a library of routines callable from almost any programming language
  - that enables communication among multiple processes
  - standardized and portable API with good implementations available for almost any kind of parallel computer
- `MPI` is large and complex
  - more than 125 functions, lot's of options and communication protocols
  - but for most practical purposes, a small subset will suffice
  - short introduction today, more when we consider specific physics
- two major versions available – check your installation for compliance
  - `MPI`-1: parallel machine management, process groups, collective operations, point-to-point operations, virtual topologies, profiling
  - `MPI`-2: dynamic process management, one-sided operations, parallel I/O, (simplistic) bindings for `C++`
- `openmpi`: currently the best open source implementation
  - well-architected, thread safe, fast, decent support from a broad community

Getting started

- compiling and linking:
  - most `MPI` implementation supply wrappers around the available compilers
    - e.g. `mpicc`, `mpic++`, `mpif77`, `mpif90`
  - it's not magic, so you can do it on your own to
    - override the system defaults (without upsetting the sysadmins...)
    - build multiple versions so you can benchmark
- staging and launching:
  - most implementations provide `mpirun` to
    - control the total number of desired processes
    - specify the hostnames of the machines to use
    - specify the mapping of processes to machines/CPUs/cores
    - establish the current working directory, if possible, for all processes
    - launch the program
  - but most installations do not permit its use; they have queuing systems instead
    - `PBS`, `LSF`, `torque`, `maui`, ...
    - specified and documented in the "welcome" package of most supercomputer centers
    - scheduling of jobs, guarantee exclusive access to your allocated machines, establish upper time limit, charge the right account for your uses

At runtime

- initializing the coöperating processes:

```
int MPI_Init(int* argc, char ***argv);
```

  - note the strange signature; see Slide 7 for an example of its use
  - some implementations – notably MPICH, the reference implementation – used command line arguments to pass information from `mpirun` to the runtime environment
  - so they need *write* access to the command line arguments to strip the extras
  - thankfully, not done any more
- must be the first MPI in your program; nothing is initialized correctly until it returns
  - if this call does not return `MPI_SUCCESS`, you should abort
- don't forget to shut everything down:

```
int MPI_Finalize(void);
```

- must be the last MPI call in your program; nothing is in usable state after it returns

Groups and communicators

- every MPI process belongs to at least one *group*
- groups have associated *communicators* that provide the context for data exchanges and synchronization among processes
- processes in a given communicator get *ranked*
  - a communicator of p processes assigns ranks 0 through $p - 1$
  - a process can discover the communicator size and its own rank by using

```
int MPI_Comm_size(MPI_Comm communicator, int* size);
int MPI_Comm_rank(MPI_Comm communicator, int* rank);
```

- the MPI runtime environment creates the *global* communicator
  - known as MPI_COMM_WORLD
  - all processes are members
- it is good practice to learn to manage your own
  - to narrow down global operations to processor subsets
  - to promote *reuse*
  - more details later

Hello world

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

int main(int argc, char* argv[]) {
   int status;
   int rank, size;

   /* initialize MPI */
   status = MPI_Init(&argc, &argv);
   if (status != MPI_SUCCESS) {
      printf("error in MPI_Init; aborting...\n");
      return status;
   }

   /* all good -- get process info and display it */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   printf("hello from %03d/%03d!\n", rank, size);

   /* shut down MPI */
   MPI_Finalize();

   return 0;
}
```

Messages

- in general, data exchanges through MPI calls involve
  - a communicator
    - specifies which processes participate in the exchange
    - resolves process ranks into processes
  - *collective* operations involve the entire communicator
  - *point-to-point* operations require the rank of the message source or destination
  - the details of the message payload
    - the address of the source buffer
    - the data type of the buffer contents
    - the number of items in the buffer
- `MPI` provides some data abstractions to
  - hide machine dependencies in the data representations to enhance portability and support heterogeneous clusters
  - support user defined data types
  - support non-contiguous data layouts

Collective operations: global reductions

- *collective* operations involve all processes in a given communicator
- the MPI version of our global reduction example uses

```
int MPI_Allreduce(
      void* send_buffer, void* recv_buffer,
      int count, MPI_Datatype datatype, MPI_Op operation,
      MPI_Comm communicator
      );
```

- example legal values for MPI_Datatype
  - C: MPI_INT, MPI_LONG, MPI_DOUBLE
  - FORTRAN: MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_COMPLEX
- legal values for MPI_Op
  - MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC
  - MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_LOR, MPI_LXOR
  - MPI_BAND, MPI_BOR, MPI_BXOR
  - MPI_REPLACE

Example reduction using MPI

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

int main(int argc, char* argv[]) {
   int status;
   int rank;
   int square, sum;

   /* initialize MPI */
   status = MPI_Init(&argc, &argv);
   if (status != MPI_SUCCESS) {
      printf("error in MPI_Init; aborting...\n");
      return status;
   }

   /* get the process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   /* form the square */
   square = rank*rank;
   /* each process contributes the square of its rank */
   MPI_Allreduce(&square, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
   /* print out the result */
   printf("%03d: sum = %d\n", rank, sum);

   /* shut down MPI */
   MPI_Finalize();

   return 0;
}
```

Point to point communication

- to send a message

```c
int MPI_Send(
      void* buffer, int count, MPI_Datatype datatype,
      int destination, int tag, MPI_Comm communicator
      );
```

- to receive a message

```c
int MPI_Recv(
      void* buffer, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm communicator
      );
```

- the `tag` enables choosing the order you may receive pending messages
- but for a given (`source`,`tag`,`communicator`) messages are received in the order they were sent
- receiving via wildcards: `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
- in *standard* communication mode, sending and receiving messages are *blocking*, so the function does not return until you can safely access the `buffer`
  - to read, free, etc.

Communication modes

- in standard mode, the specification does not explicitly mention buffering strategy
  - buffering messages would remove some of the access constraints but it requires time and storage for the multiple copies
  - portability across implementations implies conservative assumptions about the order of initiation of sends and receives to avoid deadlock
- in *ready* mode, you must post a receive before the matching send can be initiated
  - `MPI_Rsend`, `MPI_Rrecv`
- in *buffered* mode, sends can be initiated, and may complete, regardless of when the matching receive is initiate
  - `MPI_Bsend`, `MPI_Brecv`
- in *synchronous* mode, sends can be initiated regardless of whether the matching receive has been initiated, but the send will not return until the message has been received
  - `MPI_Ssend`, `MPI_Srecv`

Asynchronous communication

- there are non-blocking versions of all these

```
int MPI_Isend(
      void* buffer, int count, MPI_Datatype datatype,
      int destination, int tag,
      MPI_Comm communicator, MPI_Request* request
      );
```

  - faster, but you must take care to not access the message buffers until the messages have been delivered
  - more details later in the course, as needed
- for sends
  - standard mode: `MPI_Isend`
  - ready mode: `MPI_Irsend`
  - buffered mode: `MPI_Ibsend`
  - synchronous mode: `MPI_Issend`
- only one call for receives: `MPI_Irecv`
- extra `request` argument to check for completion of the request
  - `MPI_Test`, `MPI_Wait` and their relatives

Creating communicators and groups

- communicators and groups are intertwined
  - you cannot create a group without a communicator
  - you cannot create a communicator without a group
- the cycle is broken by `MPI_COMM_WORLD`

```
#include <mpi.h>

int main(int argc, char* argv[]) {
   /* declare a communicator and a couple of groups */
   MPI_Comm workers;
   MPI_Group world_grp, workers_grp;

   /* initialize MPI; for brevity all status checks are omitted */
   MPI_Init(&argc, &argv);

   /* get the world communicator to build its group */
   MPI_Comm_group(MPI_COMM_WORLD, &world_grp);

   /* build another group by excluding a process */
   MPI_Group_excl(world_grp, 1, 0, &workers_grp);

   /* now build a communicator out of the processes in workers_grp */
   MPI_Comm_create(MPI_COMM_WORLD, worker_grp, &workers);

   /* etc.... */

   /* shut down MPI */
   MPI_Finalize();

   return 0;
}
```

Manipulating communicators and groups

- releasing resources

```
int MPI_Group_free(MPI_Group* group);
int MPI_Comm_free(MPI_Comm* communicator);
int MPI_Comm_disconnect(MPI_Comm* communicator);
```

- you can make a new group by adding or removing processes from an existing one

```
int MPI_Group_incl(
   MPI_Group grp, int n, int* ranks, MPI_Group* new_group);
int MPI_Group_excl(
   MPI_Group grp, int n, int* ranks, MPI_Group* new_group);
```

- or by using set operations

```
int MPI_Group_union(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
int MPI_Group_intersection(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
int MPI_Group_difference(
   MPI_Group grp1, MPI_Group grp2, MPI_Group* new_group);
```

Timing

- the function

```
double MPI_Wtime();
```

returns the time in seconds from some arbitrary time in the past
  - guaranteed not to change only for the duration of the process
- you can compute the elapsed time for any program segment by making calls at the beginning and the end and computing the difference
- no guarantees about synchronized clocks among different processes
- you can compute the clock resolution by using

```
double MPI_Wtick();
```

Other collective operations

- `MPI_Scan` computes partial reductions: the p^{th} process receives the result from processes 0 through $p - 1$

```
int MPI_Scan(
      void* send_buffer, void* recv_buffer,
      int count, MPI_Datatype datatype, MPI_Op operation,
      MPI_Comm communicator
      );
```

- `MPI_Reduce` collects the result at only the given process `root`

```
int MPI_Reduce(
      void* send_buffer, void* recv_buffer,
      int count, MPI_Datatype datatype, MPI_Op operation,
      int root, MPI_Comm communicator
      );
```

- synchronization is also a global operation:

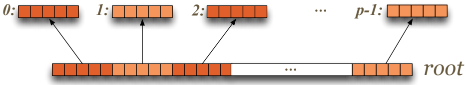
```
int MPI_Barrier(MPI_Comm communicator);
```

participating processes block at a barrier until they have all reached it

Scatter

- `MPI_Scatter` sends data from `root` to all processes

```
int MPI_Scatter(
      void* send_buffer, int send_count, MPI_Datatype send_datatype,
      void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
      int root, MPI_Comm communicator
      );
```

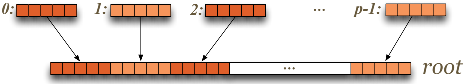


- it is as if the data in `send_buffer` were split in p segments, and the i^{th} process receives the i^{th} segment
- the `send_xxx` arguments are only meaningful for `root`; they are ignored for other processes
- the arguments `root` and `communicator` must be passed identical values by all processes

Gather

- the converse is `MPI_Gather` with `root` receiving data from all processes

```
int MPI_Gather(
        void* send_buffer, int send_count, MPI_Datatype send_datatype,
        void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
        int root, MPI_Comm communicator
        );
```



- it is as if *p* messages, one from each processes, were concatenated in rank order and placed at `recv_buffer`
- the `recv_xxx` arguments are only meaningful for `root`; they are ignored for other processes
- the arguments `root` and `communicator` must be passed identical values by all processes

Broadcasting operations

- `MPI_Alltoall` sends data from all processes to all processes

```
int MPI_Alltoall(
      void* send_buffer, int send_count, MPI_Datatype send_datatype,
      void* recv_buffer, int recv_count, MPI_Datatype recv_datatype,
      MPI_Comm communicator
      );
```

  a global scatter/gather

- use `MPI_Bcast` to send the contents of a buffer from `root` to all processes in a communicator

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

Virtual topologies

-