ACM/CS 114
Parallel algorithms for scientific applications

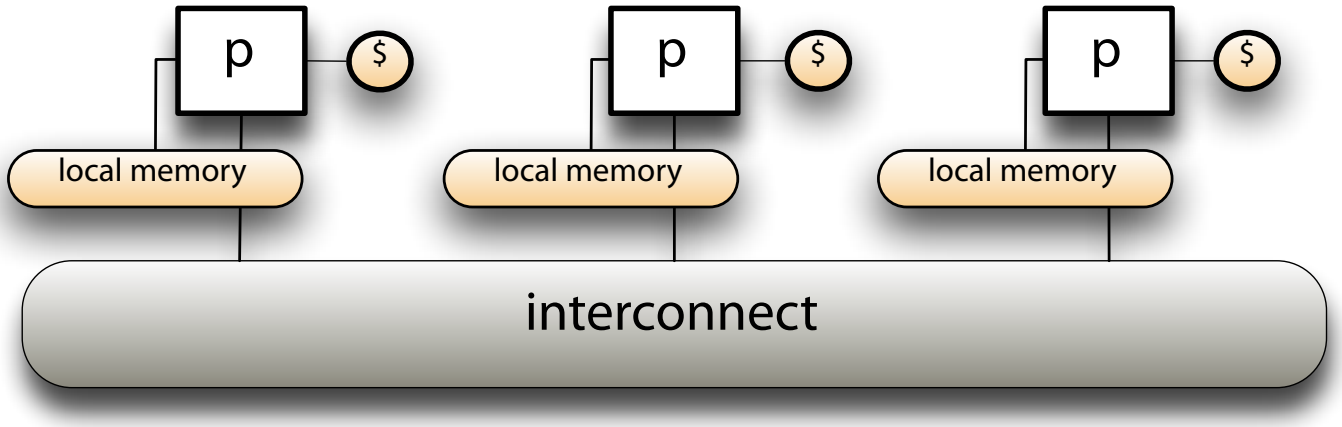
Michael A. G. Aïvázis

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:

```
1 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:

```
1 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

```
1 int MPI_Comm_size(MPI_Comm communicator, int* size);
2 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

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5     int status;
6     int rank, size;
7
8     /* initialize MPI */
9     status = MPI_Init(&argc, &argv);
10    if (status != MPI_SUCCESS) {
11       printf("error in MPI_Init; aborting...\n");
12       return status;
13    }
14
15    /* all good -- get process info and display it */
16    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17    MPI_Comm_size(MPI_COMM_WORLD, &size);
18    printf("hello from %03d/%03d!\n", rank, size);
19
20    /* shut down MPI */
21    MPI_Finalize();
22
23    return 0;
24 }
```

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

```
1 int MPI_Allreduce(
2       void* send_buffer, void* recv_buffer,
3       int count, MPI_Datatype datatype, MPI_Op operation,
4       MPI_Comm communicator
5       );
```

- 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;
}
```