Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: Message Passing

What about distributed memory?

So far we have seen three ways to create a parallel program:

1. Let the compiler do whatever it can completely automatically
2. Create threads explicitly using pthreads
3. Specify parallel sections using OpenMP

These all suffer from one significant limitation – the cooperating threads must be able to communicate through shared variables.

How can we design and run parallel programs to work when there is no shared memory available?

Message Passing

We will now consider the message passing paradigm.

- Characteristics:
  - **Locality** - each processor accesses *only* its local memory
  - **Explicit parallelism** - messages are sent and received explicitly - programmer controls all parallelism. The compiler doesn't do it.
  - **Cooperation** - every send must have a matching receive in order for the communication to take place. *Beware of deadlock!* One sided communication is possible, but doesn't really fit the pure message-passing model.
- Advantages:
  - **Hardware** - many current clusters of workstations and supercomputers fit well into the message passing model, and shared memory systems can run message passing programs as well.
  - **Functionality** - full access to parallelism. We don't have to rely on a compiler. The programmer can decide when parallelism makes sense. But this is also a disadvantage - the full burden is on the programmer! Advice: If the compiler can do it for you, let it!

– **Performance** - data locality is important - especially in a multi-level memory hierarchy which includes off-processor data. There is a chance for superlinear speedup with added cache as we talked about earlier in the course. Communication is often MUCH more expensive than computation.

Cooperating Processes

Unix programs can use `fork()` to create new processes.

The Unix system call `fork()` duplicates a process. The child is a copy of the parent - in execution at the same point, the statement after the return from `fork()`.

The return value indicates if you are child or parent.

0 is child, > 0 means parent, -1 means failure (limit reached, permission denied)

Example C program:

```
pid=fork();
if (pid) {
  parent stuff;
}
else {
  child stuff;
}
```

A more complete program that uses `fork()` along with three other system calls (`wait()`, `getpid()`, and `getppid()`) is here:

See: `/cluster/examples/forking`

Processes created using `fork()` do not share context, and must allocate shared memory explicitly, or rely on a form of message passing to communicate.

We will not consider the Unix `fork()` method of multiprocessing much, but here's an example to show you that in fact, even global variables are not shared by processes that are created with `fork()`:

See: `/cluster/examples/what_shared`

Remember that the advantage of using processes such as these instead of threads is that the processes could potentially be running on different systems. But if they are going to cooperate, they will need to communicate:

- Two processes on the same system can communicate through a named or an unnamed *pipe*.
- Two processes on different systems must communicate across a network - most commonly this is done using *sockets*.

Sockets and pipes provide only a very rudimentary interprocess communication. Each "message" sent through a pipe or across a socket has a unique sender and unique receiver and is really nothing more than a stream of bytes. The sender and receiver must add any structure to these communcations.

Here's a very simplistic example of two processes that can communicate over raw sockets. It is included mainly to show you that you don't want to be doing this if you can help it:

See: `/cluster/examples/socket`

For many applications, this primitive interface is unreasonable. We want something at a higher level. Message passing libraries have evolved to meet this need.

Message Passing Libraries

- Message passing is supported through a set of library routines. This allows programmers to avoid dealing with the hardware directly. Programmers want to concentrate on the problem they're trying to solve, not worrying about writing to special memory buffers or making TCP/IP calls or even creating sockets.
- **Examples:** P4, PVM, MPL, MPI, MPI-2, etc. MPI and PVM are the most common.
- **Core Functionality:**
  - *Process Management* - start and stop processes, query number of procs or PID.
  - *Point-to-Point Communications* - send/receive between processes
  - *Collective Communication* - broadcast, scatter, gather, synchronization
- **Terminology:**
  - *Buffering* - copy into a buffer somewhere (in library, hardware)
  - *Blocking communication* - wait for some "event" to complete a communication routine
  - *Nonblocking communication* - "post" a message and return immediately
  - *Synchronous communication* - special case of blocking - send does not return until corresponding receive completes
  - *Asynchronous communication* - pretty much nonblocking

Point-to-Point Communication

All message passing is based on the simple `send` and `receive` operations

```
P0: send(addr,len,dest,tag)

P1: receive(addr,max_len,src,tag,rec_len)
```

These are basic components in any message-passing implementation. There may be others introduced by a specific library.

- `addr` is the address of the send/receive buffer
- `len` is the length of the sent message
- `max_len` is the size of the receive buffer (to avoid overflow)
- `rec_len` is the length of the message actually received
- `dest` identifies destination of a message being sent
- `src` identifies desired sender of a message being received (or where it actually came from if "any source" is specified)
- `tag` a user-defined identifier restricting receipt

Point-to-Point Communication - Blocking

Blocking communication has simple semantics:

- `send` completes when send buffers are ready for reuse, after message received or at least copied into system buffers
- `receive` completes when the receive buffer's value is ready to use

But beware of **deadlock** *when using blocking routines!!*

```
      Proc 0          Proc 1
-------------------------------------
   bsend(to 1)     bsend(to 0)
  brecv(from 1)   brecv(from 0)
```

If both processors' send buffers cannot be copied into system buffers, or if the calls are strictly synchronous, the calls will block until the corresponding receive call is made... Neither can proceed... *deadlock...*

Possible solutions - reorder to guarantee matching send/receive pairs, or use *nonblocking routines*...

Point-to-Point Communication - Nonblocking

- `send` or `receive` calls return immediately - but how do we know when it's done? When can we use the value?
- can overlap computation and communication

- must call a `wait` routine to ensure communication has completed before destroying send buffer or using receive buffer

Example:

```
      Proc 0          Proc 1
-------------------------------------
    nbsend(to 1)    nbsend(to 0)
   nbrecv(from 1)  nbrecv(from 0)
   compute......   compute......
     waitall         waitall
    use result      use result
```

During the "compute....." phase, it's possible that the communication can be completed "in the background" while the computation proceeds, so when the "waitall" lines are reached, the program can just continue.

Deadlock is less likely but we still must be careful – the burden of avoiding deadlock is on the programmer in a message passing model of parallel computation.

Collective Communication

Some common operations don't fit well into a point-to-point communication scheme. Here, we may use *collective communication* routines.

- collective communication occurs among a group of processors
- the group can be all or a subset of the processors in a computation
- collective routines are blocking
- types of collective operations
  - *synchronization/barrier* - wait until all processors have reached a given point
  - *data movement* - broadcast (i.e. error condition, distribute read-in values), scatter/gather (exchange boundary on a finite element problem, for example), all-to-all (extreme case of scatter/gather)
  - *reductions* - collect data from all participating processors and operate on it (i.e. add, multiply, min, max)

These kinds of operations can be achieved through a series of point-to-point communication steps, but operators are often provided.

Using collective communication operators provided is generally better than trying to do it yourself. In addition to providing convenience, the message passing library can often perform the operations more efficiently by taking advantage of low-level functionality.