Message Passing with MPI

October 14, 2013

1 Message Passing

- Message passing feels like just sending data, but it is a perfectly valid means of synchronization.
- A bit more "abstract" (high level) than shared memory synchronization with locks.
- Synchronization takes place on receive and, potentially, on send.

1.1 Basics

- At the core, it is as simple as send/receive pairs.
- One process sends a message. Another receives.
- Advantage is that there is no chance for conflicts as with shared memory. But there are other concerns.

1.2 Alternatives

- Send
  - Blocking (synchronous) or non-blocking (asynchronous)? Implications? Non-blocking requires that
    buffers not be released until sent, but can reduce copying and simplify reasoning.
  - Delivered to another process or to a "mailbox?"
- Receive
  - Ability to choose message to receive?
  - Mechanism to receive from one of many sources? If a process wants to handle different messages from
    different sources (perhaps with priorities), how is this accomplished? Probe for messages, but another
    process (if in a mailbox) or thread might take it first.

1.3 Concerns

- Pairing sends/receives (synchronous). If each blocks, then ordering is extremely important to avoid deadlock.
- Data volume and frequency of messages.
- Idle times (addressed with asynchrony or buffers).
- Message selection/filtering/priority.

1.4 CSP Perspective

- Tony Hoare (Quicksort)
- Communicating Sequential Processes
- Very influential work. Simplicity in action.
- Processes are spawned and communicate only through messages.
• Output: B!e – send message e (an expression) to process B

• Input: A?x – receive a message from A into x

• Guarded Communication: Allows for non-deterministic processing. For example,

  if B?c -> ...
  [] C!e -> ...
  fi

• All communication is synchronous. This can make for some interesting programming problems.

• How does one create an asynchronous send from a synchronous send?

2 Message Passing Interface (MPI)

• A library-based approach to portable, tightly-coupled distributed programming.
  – Portable in that when used properly, the same MPI code can run on many different architectures (data types come into play here).
  – Tightly-coupled in that the intent is for processes that are working closely together to accomplish a single task. This is as opposed to something like HTTP or peer computing (examples of loosely-coupled distributed computing).

2.1 Overview

• MPI supports
  – multiple ”processes”
  – point-to-point communication
  – group communication

• Bindings exist for various languages, but the main focus has been C, C++, and Fortran.

• Format of MPI Calls
  – C names are case sensitive; Fortran names are not.
  – Programs must not declare variables or functions with names beginning with the prefix MPI, or PMPI (profiling interface).

• Communicators and Groups
  – MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
  – Most MPI routines require you to specify a communicator as an argument.
  – For now, simply use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.

• Rank
  – Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a “task ID”. Ranks are contiguous and begin at zero.
  – Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

• Error Handling
  – Most MPI routines include a return/error code parameter.
– However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).
– The standard does provide a means to override this default error handler.
– The types of errors displayed to the user are implementation dependent.

2.2 Point-to-Point

• Basic (and core) communication. Sends matched by receives.

• Send/Receive ... blocking and non-blocking options

• When does the sender continue?

• Types for different approaches/performance.
  – Standard - impl. may buffer ... sender continues when msg hits buffer or receiver
  – Buffered - buffered if space, error if not ... sender continues
  – Synchronous - blocks until actually received by a process
  – Ready - error if a receiver isn’t waiting

• Non-blocking
  – Process does not wait for completion.
  – Must check separately for completion against returned handle.
  – What is this for?

2.3 Datatypes

• Desire to send ”messages” and not just byte buffers.

• Want something ”portable.” Manually packing structures isn’t quite going to cut it.

• Library provided API for defining datatypes.

• Cumbersome (an IDL approach like CORBA would help), but sufficient.

• We’ll leave this to further study.

2.4 Group Communication

• Groups of processes can be defined from other groups.

• Communication between members of a group is accomplished through standard mechanisms.
  – Barrier.
  – Broadcast.
  – Scatter.
  – Gather.
2.5 Examples

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int num_procs, my_rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_procs); /* get communicator size */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);    /* get rank in communicator */

    /* I/O depends on implementation. Sometimes only valid in process 0. */
    printf("My rank is %d of %d\n", my_rank, num_procs);

    MPI_Finalize();

    return 0;
}
```

```c
#include <mpi.h>
#define BUFSIZE 256
#define EX_TAG 1

int main(int argc, char **argv)
{
    int num_procs, my_rank;
    char buf[BUFSIZE];
    MPI_Status stat;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    ...

    if (my_rank == 0)
    {
        MPI_Send(buf, BUFSIZE, MPI_CHAR, 1, EX_TAG, MPI_COMM_WORLD);
        ...
    }
    else if (my_rank == 1)
    {
        MPI_Recv(buf, BUFSIZE, MPI_CHAR, 0, EX_TAG, MPI_COMM_WORLD, &stat);
        ...
    }

    MPI_Finalize();

    return 0;
}
```

```c
#include <mpi.h>
#define BUFSIZE 256
#define EX_TAG_ONE 1
```
```c
#include <mpi.h>
#define BUFSIZE 256
#define EX_TAG_ONE 1
#define EX_TAG_TWO 2

int main(int argc, char **argv)
{
    int num_procs, my_rank;
    char buf[BUFSIZE];
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    ...  
    if (my_rank == 0)
    {
      MPI_Send(buf, BUFSIZE, MPI_CHAR, 2, EX_TAG_ONE, MPI_COMM_WORLD);
      ...  
    } else if (my_rank == 1)
    {
      MPI_Send(buf, BUFSIZE, MPI_CHAR, 2, EX_TAG_TWO, MPI_COMM_WORLD);
      ...  
    } else if (my_rank == 2)
    {
        /* receive message with tag one and then with tag two */
      MPI_Recv(buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_ONE, 
                MPI_COMM_WORLD, &stat);
      MPI_Recv(buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_TWO, 
                MPI_COMM_WORLD, &stat);
    }
    MPI_Finalize();

    return 0;
}
```
if (my_rank == 0) {
    /* "immediate" send */
    MPI_ISend(buf, BUFSIZE, MPI_CHAR, 1, EX_TAG_ONE, MPI_COMM_WORLD, &req);
    ...
    MPI_Test(&req, &flag, &stat);
    ...
    if (flag)
        {
            MPI_Wait(&req, &stat);
            ... cleanup ...
        }
} else if (my_rank == 1) {
    /* blocking receive */
    MPI_Recv(buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
}

MPI_Finalize();

    return 0;
}

#include <mpi.h>

#define BUFSIZE 256
#define EX_TAG_ONE 1
#define EX_TAG_TWO 2

int main(int argc, char **argv) {
    int num_procs, my_rank;
    char buf[BUFSIZE];
    MPI_Status stat;
    MPI_Request req;
    int flag;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    ...
    if (my_rank == 0) {
        /* "immediate" send */
        MPI_ISend(buf, BUFSIZE, MPI_CHAR, 1, EX_TAG_ONE, MPI_COMM_WORLD, &req);
        ...
        MPI_Test(&req, &flag, &stat);
        if (flag)
            {
                MPI_Wait(&req, &stat);
                ... cleanup ...
            }
    }
    else if (my_rank == 1) {
        /* "immediate" receive */
MPI_IRecv (buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &req);
...
MPI_Test (&req, &flag, $stat);
if (flag)
{
    MPI_Wait (&req, &stat);
    ... data available ...
}
}
MPI_Finalize ();
return 0;

2.6 Multiple Message Tags

- We want to receive messages from multiple sources, but prioritize based on tag.
- Can we do this?
- 1: Waits for a message of each type. Looping requires wait on each iteration.
- 2: Pending receives for message of each type. Must handle each message. Looping creates multiple pending
takes. Ugh.
- 3: Probe for a message. If found, receive it. Looping continues to look for messages handling no more than
one each iteration.

/* want to give priority to messages with tag EX_TAG_ONE .. how? */

/* 1 */
MPI_Recv (buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_ONE, MPI_COMM_WORLD);
MPI_Recv (buf2, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_TWO, MPI_COMM_WORLD);

/* 2 */
MPI_IRecv (buf, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_ONE,
          MPI_COMM_WORLD, &req1);
MPI_IRecv (buf2, BUFSIZE, MPI_CHAR, MPI_ANY_SOURCE, EX_TAG_TWO,
          MPI_COMM_WORLD, &req2);

/* determine which arrived and handle that message */
MPI_Test (&req1, &flag, $stat);
if (flag)
{
    MPI_Wait (&req, &stat);
    ... data available ...
}
MPI_Test (&req2, &flag, $stat);
if (flag)
{
    MPI_Wait (&req, &stat);
    ... data available ...
}

/* 3 */
int conditional_recv_tag (char *buf, int size, int tag, MPI_Status *stat)
{
    MPI_IProbe (MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &flag, stat);
    if (flag)
    {

MPI_Recv(buf, BUFSIZE, MPLCHAR, stat->MPI_SOURCE, tag, MPI_COMM_WORLD, stat);
}
return flag;
}

...
if (conditional_recv_tag(buf, BUFSIZE, EX_TAG_ONE, &stat))
{ ... handle message ... }
else if (conditional_recv_tag(buf, BUFSIZE, EX_TAG_TWO, &stat))
{ ... handle message ... }

2.7 Simple Exercise
• Simple prime test.
• Given a number, check if it is prime via simple division.
• Worker/Bag-of-tasks model. Manager creates a "bag" with N partitions of the "search space." (Granularity issue. What is a reasonable size for a partition?)
• Workers (fewer than N) repeatedly process partitions.
• What is the message passing scheme?