MPI bindings in Haskell

Exploring the design space

Outline

- Overview of MPI and demo.
- The Haskell FFI.
- C2HS, a tool for helping to write bindings to C.
- A first stab.
- A second stab.
- A tricky case: non-blocking receive.
- The future and beyond.

MPI - Message Passing Interface

- A protocol for distributed parallel communication.
- *De facto* standard in distributed high performance computing.
- Exposed to the world via a programming language API.
- Official support for C, C++ and Fortran.
- Other languages usually bind to one of the official APIs.
- Many mature implementations (OpenMPI, MPICH, LAM/MPI, various hardware vendors have their own/modified versions).

MPI - Message Passing Interface

- Execution model influenced by unix processes.
- Each parallel task is called a “process”. Normally each process is run on a different CPU (core), but this is not required.
- Each process has its own address space.
- A job is usually launched with mpirun/mpiexec, which forks off a number of processes and distributes them across CPUs.
- It is typical to employ a Single Program Multiple Data (SPMD) style of programming, but it is not required.
- A process can be multi-threaded.

MPI - Message Passing Interface

- Point to point communication: (e.g.) MPI_Send, MPI_recv.
- Collective communication: (e.g.) MPI_Bcast, MPI_Alltoall, MPI_Gather, MPI_Scatter.
- Synchronisation: (e.g.) MPI_barrier.
- Data serialisation: (e.g.) MPI_pack
- Parallel I/O.

A taste of MPI in Haskell

```
main :: IO ()
main = mpi $ do
   rank <- commRank commWorld
   size <- commSize commWorld
   if (rank /= root)
      then send (msg rank) root tag commWorld
      else do
         forM_ [1..size-1] $ \sender -> do
         (_status, result) <- recv (toRank sender) tag commWorld
         putStrLn result

msg :: Rank -> String
msg r = "Greetings from process " ++ show r ++ "!"

root :: Rank
root = toRank 0

tag :: Tag
tag = toTag ()
```

Compiling, then running a 10 process job

```
$ ghc --make Greetings.hs
[1 of 1] Compiling Main             ( Greetings.hs, Greetings.o )
Linking Greetings ...

$ mpirun -np 10 ./Greetings
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
Greetings from process 8!
Greetings from process 9!
```

Getting our hands dirty with the FFI

MPI_Send prototype in C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Direct foreign wrapper in Haskell:

```
foreign import ccall unsafe "MPI_Send"
  send :: Ptr () -> CInt -> Ptr () -> CInt -> CInt -> Ptr () -> IO CInt
```

Automation to the rescue

- Writing the direct foreign wrapper code is tedious and error prone.
- Better to get a tool to do all the hard work.
- We have a few choices in Haskell: Green Card, hsc2hs, c2hs.
- c2hs processes C header files, so it remains in sync with the C API.

Automation to the rescue

```
#include <mpi.h>

{# context prefix = "MPI" #}

send = {# call unsafe Send as send_ #}

{# fun unsafe Comm_rank as ^ { id `Comm', alloca- `Int' peekIntConv* } -> `Int' #}
```

Automation to the rescue

```
data Status =
   Status
   { status_source :: Int
   , status_tag :: Int
   , status_error :: Int
   , status_count :: Int
   , status_cancelled :: Int
   }
   deriving (Eq, Ord, Show)

{# pointer *Status as StatusPtr -> Status #}

instance Storable Status where
  sizeOf _ = {#sizeof MPI_Status #}
  alignment _ = 4
  peek p = Status
    <$> liftM cIntConv ({#get MPI_Status->MPI_SOURCE #} p)
    <*> liftM cIntConv ({#get MPI_Status->MPI_TAG #} p)
    <*> liftM cIntConv ({#get MPI_Status->MPI_ERROR #} p)
    <*> liftM cIntConv ({#get MPI_Status->_count #} p)
    <*> liftM cIntConv ({#get MPI_Status->_cancelled #} p)
  poke p x = do
    {#set MPI_Status.MPI_SOURCE #} p (cIntConv $ status_source x)
    {#set MPI_Status.MPI_SOURCE #} p (cIntConv $ status_tag x)
    {#set MPI_Status.MPI_ERROR #} p (cIntConv $ status_error x)
    {#set MPI_Status._count #} p (cIntConv $ status_count x)
    {#set MPI_Status._cancelled #} p (cIntConv $ status_cancelled x)
```

A first stab at a more usable binding to MPI_Send

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Haskell:

```
send :: (Storable e, MArray a e IO) => a Int e -> Rank -> Tag -> Comm -> IO Int
```

A first stab at a more usable binding to MPI_Send

- Apes the C interface.
- Not clear which kind of array we want to use, so we go for the MArray interface. Any mutable array will do.
- Big problem is all array elements must be Storable, which in turn means they must all have a fixed memory size.
- Hard to send an arbitrary data structure.

A second stab at MPI_Send

```
send :: Serialize msg => msg -> Rank -> Tag -> Comm -> IO ()
send = sendBS . encode

sendBS :: ByteString -> Rank -> Tag -> Comm -> IO ()
sendBS bs rank tag comm = do
   let cRank  = fromRank rank
       cTag   = fromTag tag
       cCount = cIntConv $ length bs
   unsafeUseAsCString bs $ \cString ->
       checkError $ Internal.send (castPtr cString) cCount byte
                                  cRank cTag comm
```

Now we can send and recv interesting things!

```
type Msg = (Bool, Int, String, [()])

msg :: Msg
msg = (True, 12, "fred", [(), (), ()])

main :: IO ()
main = mpi $ do
   size <- commSize commWorld
   when (size >= 2) $ do
      rank <- commRank commWorld
      when (rank == sender) $
         send msg receiver tag commWorld
      when (rank == receiver) $ do
         (_status, result) <- recv sender tag commWorld
         print (result :: Msg)
```

Tricky case: non-blocking recv

- Receiver doesn't wait for message completion. Can continue working while message is delivered.

```
int MPI_Irecv(void *, int, MPI_Datatype, int, int,
MPI_Comm, MPI_Request *)
```

- Result is a MPI_Request type which receiver can poll to see if message is available.

- Can you think of a problem for a high-level Haskell binding?

Problem: how much memory to allocate on Irecv?

- The receiver doesn't know how big the message is going to be.
- This is trouble if we are going to send and receive arbitrary Haskell data structures.
- Can't make receiver wait for the full message - i.e. block - that would defeat the purpose of Irecv.
- Can you think of a solution?

One possible solution: futures!

```
data Future a =
   Future
   { futureThread :: ThreadId
   , futureStatus :: MVar Status
   , futureVal :: MVar a
   }

waitFuture :: Future a -> IO a
waitFuture = readMVar . futureVal

pollFuture :: Future a -> IO (Maybe a)
pollFuture = tryTakeMVar . futureVal

cancelFuture :: Future a -> IO ()
cancelFuture = killThread . futureThread
```

One possible solution: futures!

```
recvFuture :: Serialize msg => Rank -> Tag -> Comm -> IO (Future msg)
recvFuture rank tag comm = do
   valRef <- newEmptyMVar
   statusRef <- newEmptyMVar
   threadId <- forkIO $ do
      (status, msg) <- recv rank tag comm
      putMVar valRef msg
      putMVar statusRef status
   return $ Future
            { futureThread = threadId
            , futureStatus = statusRef
            , futureVal = valRef }
```

One possible solution: futures!

```
type Msg = [Int]

msg :: Msg
msg = [1..5000000]

main :: IO ()
main = mpi $ do
   rank <- commRank commWorld
   when (rank == sender) $
      send msg receiver tag commWorld
   when (rank == receiver) $ do
      future <- recvFuture sender tag commWorld
      busyWork 1000
      result <- waitFuture future
      print (length (result :: Msg))
```

The future and beyond

- Probably end up with three levels in the Haskell bindings:

    1. Control.Parallel.MPI.Internal
    2. Control.Parallel.MPI.Array
    3. Control.Parallel.MPI

- Ultimately want to build more abstract operations on top of the bindings:
    - Parallel strategies.
    - MapReduce.
    - Write some parallel applications (the fun part).