Sistemas distribuidos – Message Passing Interface

Sistemas Operativos

Primer cuatrimestre de 2018

Sistemas distribuidos

- N procesos corriendo en M equipos físicamente separados.
- Se acabó la memoria compartida en el caso general.

  Podría haberla en ciertos casos particulares, pero eso no cambia la cuestión de fondo.
- Los procesos sólo pueden intercambiar *mensajes*.
- ¿De qué hablamos cuando hablamos de mensajes?

Pasaje de mensajes en el mundo real

- ¿Qué primitivas necesitamos?
- ¿Cómo formalizamos la semántica de cada una?
- ¿Cómo representamos los datos en los mensajes?
- ¿Dónde están los mensajes que están "en vuelo"?

- Enorme diversidad de hardware y software de base.
- Necesitamos elegir y adoptar *middleware* apropiado.
- En la materia usaremos MPI (Message Passing Interface).

¿Qué es MPI?

- Es un libro.
- Es un estándar.
- Es una interfase.
- Es una especificación.

- No es una biblioteca.
- No es una implementación.
- No es un compilador especial ni “paralelizador”.

- Es una especificación estándar de interfase para creadores, desarrolladores y usuarios de bibliotecas de pasaje de mensajes.

Implementaciones de MPI

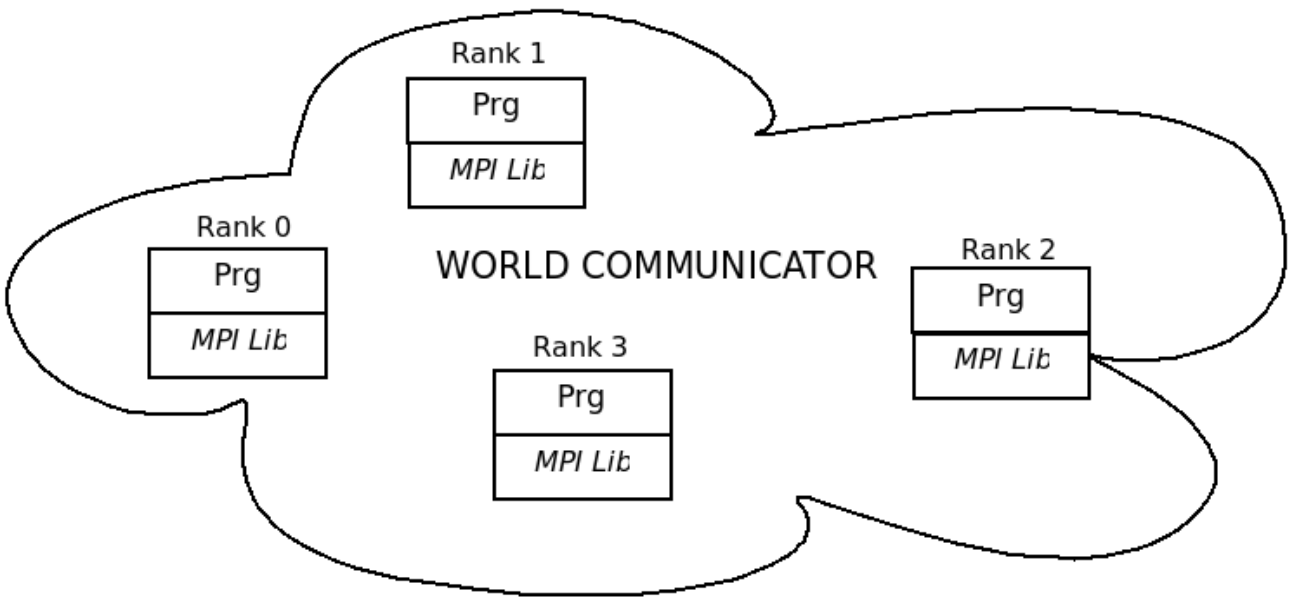
- Open source de propósito general
  - MPICH2
  - OpenMPI
- Open source de propósito específico
  - MVAPICH2 (redes InfiniBand)
- Comerciales y "vendor implementations"
  - HP
  - IBM
  - Microsoft
  - Muchas otras ad-hoc para su hardware particular

Si respetamos a rajatabla el MPI Standard, nuestro código debería funcionar correctamente en cualquier implementación.

Nos adentramos en MPI

Communicators y ranks

- En MPI, *rank* es el número que identifica a cada proceso.
- Un communicator es una organización lógica que define cuáles procesos pueden comunicarse con cuáles otros.
- Permite separar los procesos en grupos y/o armar topologías (lógicas) convenientes para el patrón de comunicaciones.
- Casi todas las primitivas reciben uno como parámetro.
- El communicator `MPI_COMM_WORLD` ya viene creado "gratis", e incluye a todos los procesos, con *ranks* que van de 0 a $N - 1$.



Tipos de primitivas

- Básicas (inicialización, finalización, quién-soy, etc).
  - `Init()`, `Finalize()`, ...
  - `Comm_rank()`, `Comm_size()`, ...
- *Point-to-point communication*
  - `Send()`, `Recv()`, ...
  - `Isend()`, `Irecv()`, ...
- *Collective communication*
  - `Barrier()`, `Bcast()`, ...
  - `Scatter()`, `Gather()`, ...
- Otras más avanzadas
  - Tipos de datos compuestos
  - RMA (acceso a memoria remota)
  - Creación dinámica de procesos...

Cómo compilar un programa que usa MPI

- Usando el script `mpicc` (o `mpic++` para C++).
- No son compiladores especiales, sino simples "wrappers".
- En nuestro caso llamarán a `gcc` y a `g++`, respectivamente.
- Se ocupan de agregar todas las opciones de línea de comando necesarias para linkear con las bibliotecas adecuadas, etc.

Cómo ejecutar un programa que usa MPI

- Usando el programa `mpiexec`.
- Ejemplo: `mpiexec -np 8 ./holamundo_distribuido`
- Con `-np` (o simplemente `-n`) se ajusta la cantidad de procesos.
- Por defecto se corre todo en localhost.
- Hay opciones para indicar en qué hosts ejecutar, cuántos procesos ejecutar en cada host, etc.

Bloqueante vs. no bloqueante en SDs

- ¿Qué significa **exactamente** "bloqueante"?
- ¿Qué significa **exactamente** "no bloqueante"?
- En sistemas distribuidos estas nociones se complican.
- ¿Por qué se complican?
- ¿Qué sucede entre un send "acá" y un receive "allá"? ¿Por dónde pasa el mensaje?

Ejemplos:

- **MPI_Send:** No devuelve hasta que se pueda usar el buffer de envío. ¿Qué pasa si lo que se envía entra en un buffer interno de MPI? ¿Qué pasa sino?.
- **MPI_Ssend:** No devolverá hasta no tener seguridad de recepción del buffer del otro lado.
- **MPI_Isend:** No bloqueante, pero no necesariamente asincrónico. No se puede reusar el buffer hasta no tener seguridad de que el mensaje haya sido copiado al buffer interno o recibido.
- **MPI_Issend:** No bloqueante. No se puede reusar el buffer hasta no tener seguridad de que el mensaje haya sido recibido.