Otra arquitectura paralela posible es una en que cada CPU tiene su propia memoria local privada, y, consecuentemente, su propio espacio físico de direcciones —**multicomputador** o **sistema de memoria distribuida**:

- la CPU tiene acceso a su memoria mediante las instrucciones LOAD y STORE
- ninguna otra CPU en el sistema tiene acceso a esta memoria
- p.ej., el cluster de Google

Las CPUs de un multicomputador no se pueden comunicar escribiendo y leyendo una memoria común (como en un multiprocesador)

… lo hacen pasándose mensajes a través de una red de interconexión

Implicaciones para el software debido a la ausencia de memoria compartida por hardware:

- si la CPU 0 descubre —de alguna manera— que parte de los datos que necesita están en la memoria de la CPU 1

  … le envía un mensaje a la CPU 1 solicitándole una copia de los datos

  … y se bloquea a la espera de que la solicitud sea respondida

- cuando la CPU 1 recibe el mensaje, el software en esa CPU primero analiza el mensaje y luego envía los datos solicitados
- cuando el mensaje de respuesta llega a la CPU 0, el software en esta CPU es desbloqueado y continúa ejecutando

La comunicación entre los procesos normalmente usa operaciones básicas de software (o *primitvas* de software):

- **`send`**
- **`receive`**

… por lo que el software es más complicado que en un multiprocesador (en que simplemente usa LOAD y STORE)

Pero …

… multicomputadores grandes, de miles de CPUs, son más simples y baratos de construir que multiprocesadores con el mismo número de CPUs:

- además, la competición por memoria en un multiprocesador puede afectar seriamente su desempeño

Cada nodo en un multicomputador:

- una o unas pocas CPUs
- memoria RAM (compartida por las CPUs en ese nodo solamente)
- disco y otros dispositivos de i/o
- un procesador de comunicaciones

Los procesadores de comunicaciones están conectados a través de una red de interconexión de alta velocidad:

- cuando una aplicación ejecuta la primitiva **send** (en alguna CPU)

  … el procesador de comunicaciones es notificado y transmite un bloque de datos al nodo destino

Las redes de interconexión, para multicomputadores (para conectar las CPUs entre ellas)

… y para multiprocesadores (para conectar las CPUs a los módulos de memoria), son similares:

- en el fondo, ambas usan paso de mensajes
- en una máquina con una sola CPU, cuando el procesador quiere leer o escribir una palabra, coloca en 1 ciertas líneas del bus (i.e. envía un *request*) y espera la respuesta
- en multiprocesadores grandes, la comunicación entre una CPU y memoria remota consiste en que la CPU envía un mensaje explícito—un paquete— a la memoria solicitando algunos datos y la memoria envía de vuelta un paquete de respuesta

La programación de un multicomputador requiere software especial —librerías— para manejar la comunicación y sincronización entre procesos

En un sistema de paso de mensajes, dos o más procesos corren independientemente:

- paso de mensajes sincrónico —el emisor permanece bloqueado hasta que el receptor haya recibido el mensaje; cuando el emisor es reanudado, sabe que el mensaje fue recibido correctamente
- paso de mensajes basado en *buffers* —el emisor puede continuar inmediatamente después de enviar el mensaje, independientemente del estado del receptor; pero no tiene garantía de que el mensaje fue recibido correctamente

¿Cómo escribimos programas para arquitecturas de memoria distribuida?

Hay que definir una interfaz con la *red de interconexión*

Definiremos **operaciones de red** que incluyan sincronización:

- operaciones básicas de *paso de mensajes*

Los procesos compartirán **canales**:

- rutas de comunicación entre procesos
- un canal es una abstracción de una red

Un canal es una cola de mensajes enviados pero aún no recibidos

Declaración de un canal:

channel <ch>(<tp_1> <id_1>, …, <tp_n> <id_n>)

- <ch> es el nombre del canal
- <tp_i> y <id_i> son los tipos (obligatorios) y los nombres (opcionales) de los campos del mensaje transmitido

P.ej.,

- `channel input(char)`
- `channel diskAccess(int cylinder, int block, int count, char* buffer)`
- `channel[n] result(int)` —arreglo de canales

send ch(<$expr_1$>, …, <$expr_n$>)

Un proceso envía un mensaje al canal ch ejecutando
send ch(<$expr_1$>, …, <$expr_n$>)

- <$expr_i$> son expresiones cuyos tipos deben corresponder con los tipos de los campos en la declaración de ch

El efecto de ejecutar **send** es

- evaluar las expresiones
- agregar un mensaje con estos valores al final de la cola asociada con el canal **ch**

La cola es conceptualmente ilimitada,

… por lo tanto la ejecución de **send** no produce demora:

- **send** es una operación **no bloqueante**

`receive ch(<var1>, …, <varn>)`

Un proceso recibe un mensaje desde el canal `ch` ejecutando `receive ch(<var1>, …, <varn>)`

- `<vari>` son variables cuyos tipos deben corresponder con los tipos de los campos en la declaración de `ch`

El efecto de ejecutar **`receive`**:

- el proceso se suspende hasta que haya al menos un mensaje en la cola del canal **`ch`**
- entonces el proceso saca el mensaje que está al comienzo de la cola y asigna sus campos a los **`<vari>`**

`receive` es una operación **bloqueante**:

- el proceso no necesita usar espera ocupada

Los canales se comportan bien

El acceso a un canal es ininterrumpible (atómico)

La entrega de un mensaje es confiable y libre de errores

Todo mensaje que es enviado a un canal es entregado, y sin ser corrompido

Los canales son colas FIFO:

- los mensajes son recibidos en el mismo orden en que fueron agregados al canal

Ejemplo: Un proceso recibe caracteres, los agrupa en líneas y envía las líneas

```
channel input(char), output(char[MAXLN])

proc CharToLine:
   char[] line = new char[MAXLN]
   int i = 0
   while (true):
       receive input(line[i])
       while (line[i] != CR && i < MAXLN-1):
           i++
           receive input(line[i])
       line[i] = EOL
       send output(line)
       i = 0
```

(`CharToLine` es un proceso de tipo **filtro**:

- recibe datos por canales de entrada, los procesa, y envía el resultado del procesamiento por canales de salida)

Un proceso puede querer hacer algo si no hay mensajes disponibles

`empty(<ch>)`

- devuelve `true` si el canal <*ch*> no contiene mensajes
- en otro caso, devuelve `false`

Precauciones:

- si un proceso llama a **`empty`** y obtiene `true`, puede que haya mensajes en la cola cuando el proceso continúe su ejecución
- si un proceso llama a **`empty`** y obtiene `false`, puede que no haya mensajes en la cola cuando el proceso trate de recibir uno

Ejemplo: Ordenar *n* números

```
proc Sort:
```
recibir todos los números desde el canal de input
ordenar los números
enviar los números ordenados al canal de output

Como **`receive`** es bloqueante, **`Sort`** debe poder determinar cuándo ha recibido todos los números:

- incluir *n* o un centinela en los datos de entrada

(`Sort` también es un proceso de tipo filtro)

Otra posibilidad:
Ordenación por mezclas sucesivas

Repetidamente, y en paralelo, mezclar dos listas ordenadas en una lista ordenada más larga

La red es construida a partir de procesos —filtros— **`Merge`**:

- cada filtro recibe valores desde dos secuencias de entrada ordenadas, in1 e in2, y produce una secuencia de salida ordenada, out
- en particular, cada filtro repetidamente compara los dos próximos valores recibidos desde in1 e in2, y envía el menor a out

Código de cada proceso Merge y declaración de canales

```
channel in1(int), in2(int), out(int)

proc Merge:
  int v1, v2
  receive in1(v1); receive in2(v2)
  while (v1 != EOS && v2 != EOS):
      if (v1 <= v2):
          send out(v1); receive in1(v1)
      else:
          send out(v2); receive in2(v2)
  if (v1 == EOS):
      while (v2 != EOS):
          send out(v2); receive in2(v2)
  else:
      while (v1 != EOS):
          send out(v1); receive in1(v1)
  send out(EOS)
```

Formamos una red de ordenación con procesos Merge y canales de entrada y salida

Disponemos los procesos y los canales en la forma de un árbol binario:

- empleamos n–1 procesos **Merge**
- el árbol tiene $\log_2 n$ niveles

Los canales deben ser compartidos:

- el canal de salida de un proceso debe ser el mismo que uno de los canales de entrada de otro proceso

Dos ventajas de los procesos tipo filtro: Interconectividad y reemplazabilidad

Pueden ser interconectados de diversas formas:

- sólo se requiere que la salida de un filtro cumpla las suposiciones de entrada de otro filtro

Si los comportamientos de entrada y salida observables externamente son los mismos, podemos reemplazar un filtro —o red de filtros— por un proceso o una red diferente:

- p.ej., podemos reemplazar el proceso Sort por una red de procesos **`Merge`** más un proceso que distribuya los valores de entrada a la red

Pares interactuantes: Otro modelo
El problema de intercambiar valores

Hay *n* procesos,

… cada uno tiene un valor local *v*,

… y el objetivo es que cada proceso sepa cuál es el menor y cuál es el mayor de los *n* valores

Tres posibles patrones de comunicación:

- centralizado (proceso coordinador)
- simétrico
- anillo

Solución centralizada

Un proceso (coordinador) junta los *n* valores,

… calcula el mínimo y el máximo de ellos,

… y envía los resultados a los otros procesos

Usa 2(*n*–1) mensajes (sólo *n*, si hay *broadcast*)

```
channel values(int)
channel[n] results(int, int)

process P[0]: —proceso coordinador
   int v —lo suponemos inicializado
   int new, smallest = v, largest = v
   for (j = 1 … n):
       receive values(new)
       if (new < smallest):
           smallest = new
       if (new > largest):
           largest = new
   for (j = 1 … n):
       send results[j](smallest, largest)

process P[k = 1 … n-1]:
   int v —lo suponemos inicializado
   int smallest, largest
   send values(v)
   receive results[k](smallest, largest)
```

Solución simétrica

Cada proceso ejecuta el mismo algoritmo:

- primero envía su valor local a todos los otros,
- … y luego calcula el mínimo y el máximo de los *n* valores, a medida que va recibiendo los valores de los otros procesos

Usa *n*(*n*–1) mensajes (sólo *n*, si hay *broadcast*)

```
channel[n] values(int)

process P[k = 0 … n-1]:
    int v —lo suponemos inicializado
    int new, smallest = v, largest = v
    for (j = 0 … n):
        send values[j](v)
    for (j = 1 … n):
        receive values[k](new)
        if (new < smallest):
            smallest = new
        if (new > largest):
            largest = new
```

Solución basada en un anillo lógico

Cada proceso recibe mensajes de su predecesor y envía mensajes a su sucesor

Cada proceso, primero, recibe dos valores, calcula el mínimo y el máximo de estos junto a su propio valor, y envía los resultados

… luego, recibe el mínimo y máximo globales y los envía

Usa 2(*n*–1) mensajes

```
channel[n] values(int smlst, int lrgst)

process P[0]:
    int v —lo suponemos inicializado
    int smlst = v, lrgst = v
    send values[1](smlst, lrgst)
    receive values[0](smlst, lrgst)
    send values[1](smlst, lrgst)

process P[k = 1 … n-1]:
    int v —lo suponemos inicializado
    int smlst, largest
    receive values[k](smlst, lrgst)
    if (v < smlst):
        smlst = v
    if (v > lrgst):
        lrgst = v
    send values[k+1](smlst, lrgst)
    receive values[k](smlst, lrgst)
    if (k < n-1):
        send values[k+1](smlst, lrgst)
```