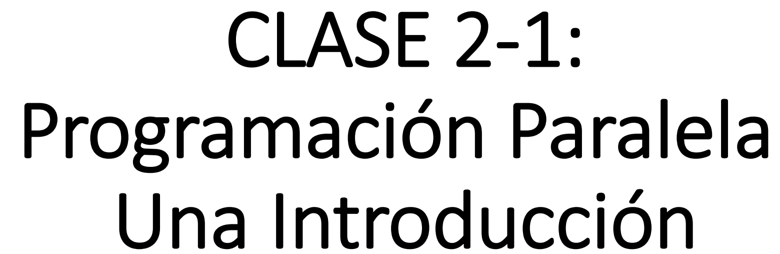


Instructor Rina Surós

29 de Septiembre 2022

Facultad de Ingeniería. UDELAR

Montevideo

Paradigmas de Programación Paralela

La estrategia de proyección a la máquina requiere de un paradigma de programación

Un **paradigma de programación paralela,** es un conjunto de métodos de diseño de códigos que nos ayuda a decidir en que forma se organiza el código. La idea es establecer una estructura de control para los algoritmos que programemos, entonces podemos decir que son esqueletos de código que nos permiten proyectar un código sobre un computador paralelo.

Los paradigmas se basan en los siguientes criterios:

- Propiedades del proceso (estructura, topología y ejecución).
- Propiedades de interacción.
- Propiedades de los datos (división y localización).

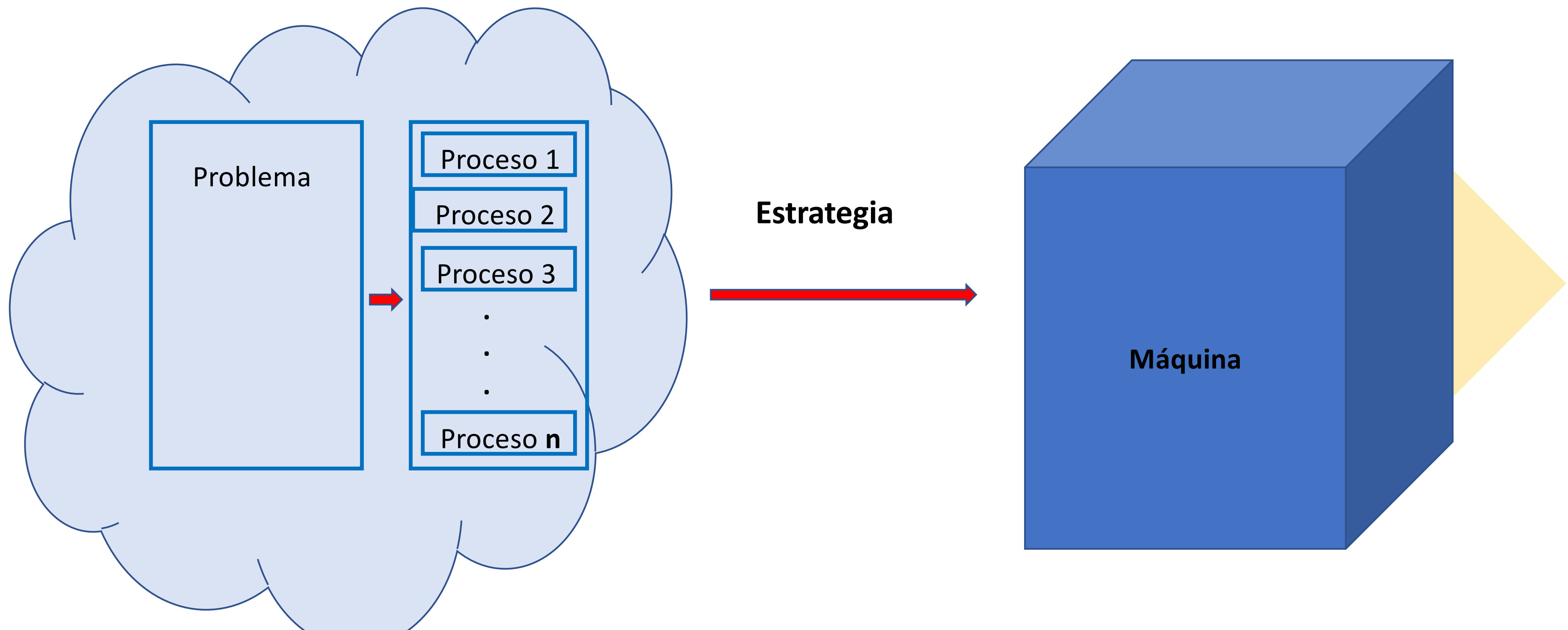
Para introducir los paradigmas de programación paralela, debemos conocer algunos conceptos:

- Comunicación por paso de mensajes
- Procesos Maestro y Esclavo
- Procesos Emisor y Receptor
- Procesos Síncronos y Asíncronos
- Tiempo Secuencial y Tiempo Paralelo
- Aceleración y Eficiencia de Procesos
- Ley de Amdahl
- Granularidad
- Latencia

En este capítulo comenzaremos a verlos

Preparando el código paralelo

Supongamos que ya hemos analizado el algoritmo y decidimos que vamos a descomponer nuestro problema en **n** procesos independientes.



Los paradigmas más comunes son:

- ***Desconposición iterativa***: Aplicaciones basadas en la ejecución de un lazo donde cada iteración se puede realizar de forma independiente.
- ***Paralelismo algorítmico***: El cual se centra en paralelizar el flujo de los datos de entrada.
- ***Descomposición geométrica***: El dominio del problema se divide en pequeños subdominios y cada procesador ejecuta el algoritmo en la parte del subdominio que le corresponde.
- ***Descomposición especulativa***: Se intentan N técnicas de solución simultáneamente, y (N- l) de ellas se eliminan tan pronto como una de ellas devuelve una respuesta correcta.

•***Descomposición funcional***: La aplicación se divide en distintas fases, y cada fase ejecuta una parte diferente del algoritmo para resolver el problema.

•***Maestro/Esclavo***: El proceso maestro es el responsable de descomponer el problema entre sus procesos esclavos y, posteriormente, recoger los resultados que le envían los esclavos para ordenarlos y obtener el resultado final.

•***SPMD (Single Program Multiple Data)***: En este paradigma cada procesador ejecuta el mismo código pero sobre distintas partes de los datos.

•***Descomposición recursiva o divide y vencerás***: El problema se divide en subproblemas que se resuelven de forma independiente para, posteriormente, combinar sus resultados parciales y obtener el resultado final.

Paradigma Maestro-Esclavo

Es un nombre de este paradigma es bastante desagradables pero es así como lo encontrarán en la literatura. Esta técnica de procesamiento paralelo es sencilla pero muy eficiente. Tenemos dos tipos de procesos

Proceso Maestro: Proceso principal, que controla la ejecución.

Procesos Esclavos Uno o más procesos secundarios que ejecutan los cálculos.

Veamos primero un ejemplo de la vida real donde las cosas suceden de esta forma: cursos a distancia

Clases a distancia modalidad Síncrona

Modalidad síncrona de enseñanza a distancia: el profesor crea una sala zoom donde se conectan todos los estudiantes de un mismo curso, es el inicio de la clase. El profesor desarrolla su clase en vivo, al final todos se desconectan simultáneamente, fin de la clase. **El profesor coordina el inicio y el fin de la clase.**

Clases a distancia modalidad Asíncrona

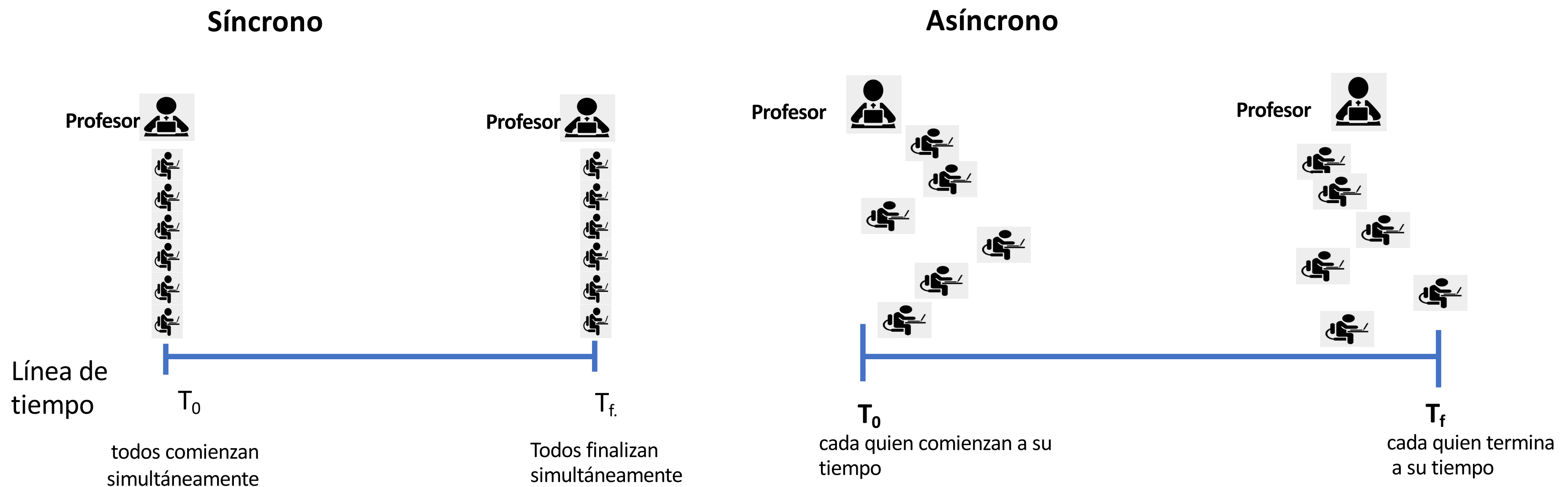
En la modalidad asíncrona de enseñanza a distancia, el profesor sube el material de un curso (Moodle por ejemplo). Eso marca el inicio del curso. Cada estudiante hace el curso a su tiempo, todos hacen el mismo curso, incluyendo tareas y evaluaciones. Se dan las fechas de inicio y fin, cuando el último estudiante termina se cierra el curso.

El profesor coordina el inicio y el fin del curso. Cada estudiante trabaja a su tiempo

Profesor

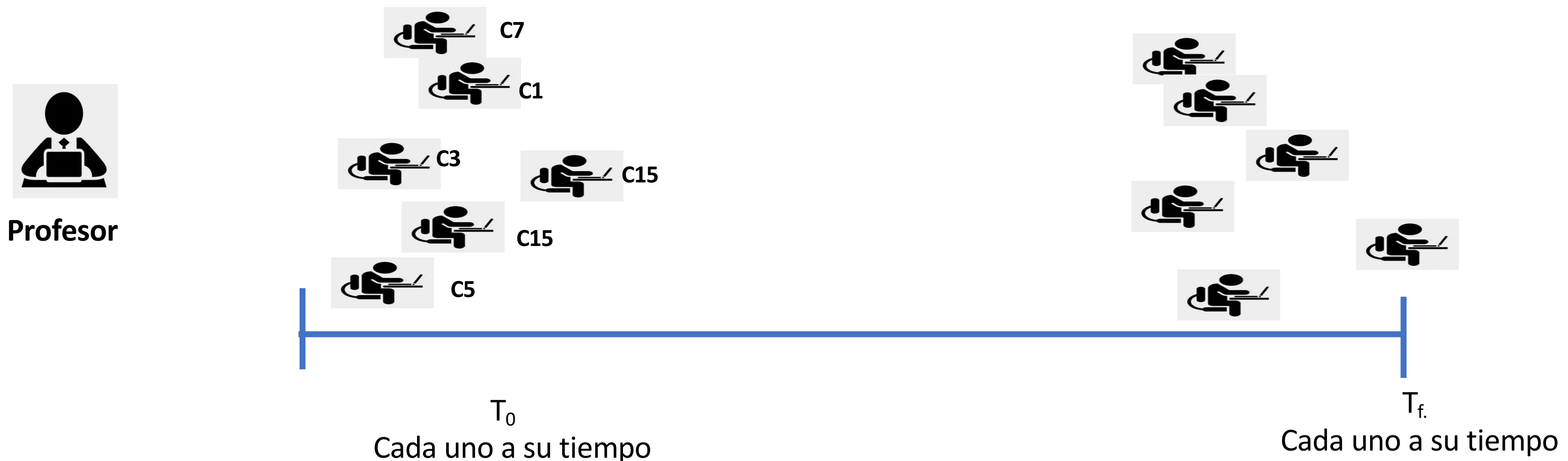
Ejemplo : clases a distancia donde el profesor es a su vez estudiante

En esta modalidad, el profesor coordina la actividad docente y el mismo forma parte como estudiante del curso. Puede ser síncrono o asíncrono. **El profesor coordina el inicio y el fin del curso y forma parte como cursante.**



Ejemplo : clases de formación continua a distancia

En esta modalidad, el profesor abre una serie de cursos –distintos- de formación continua a distancia, todos completamente independientes entre si. Inicia los cursos y publica el plazo para finalizar. Cada estudiante selecciona el curso que desea seguir y trabaja independientemente de todos los demás. El **profesor coordina el inicio y el fin de los cursos.**



Comunicación por Paso de Mensajes

- En todos los ejemplos descritos en los ejemplos se establecen procesos de comunicación para el control de los cursos.
- La comunicación entre el maestro y los procesos en ejecución se hace por paso de mensajes

Procesos Esclavos Síncronos: la ejecuciones inician y terminan simultáneamente

Procesos Esclavos Asíncronos: las ejecuciones inician y terminan en instantes de tiempo diferentes. Si el Proceso Maestro procesa datos entonces es esclavo también.

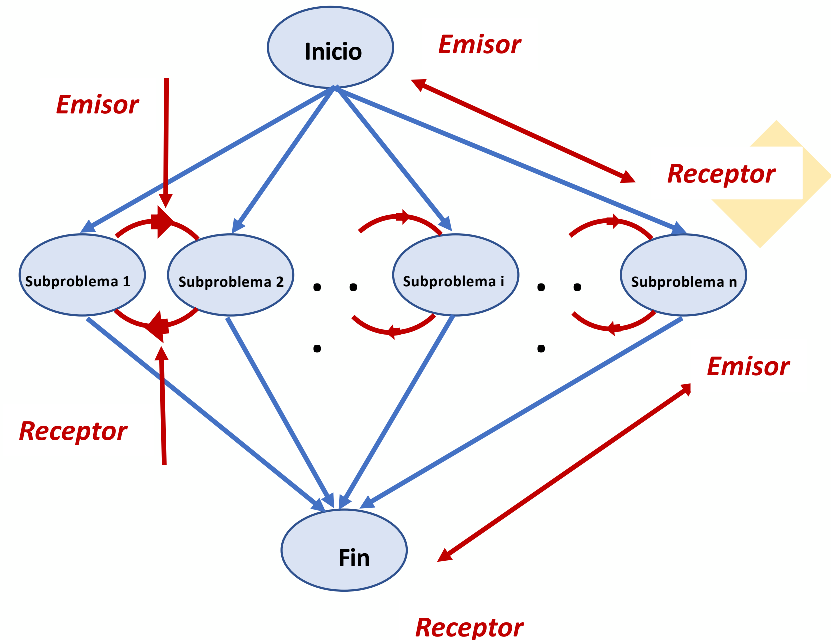
El Proceso Maestro puede asignar datos de forma estática o dinámica.

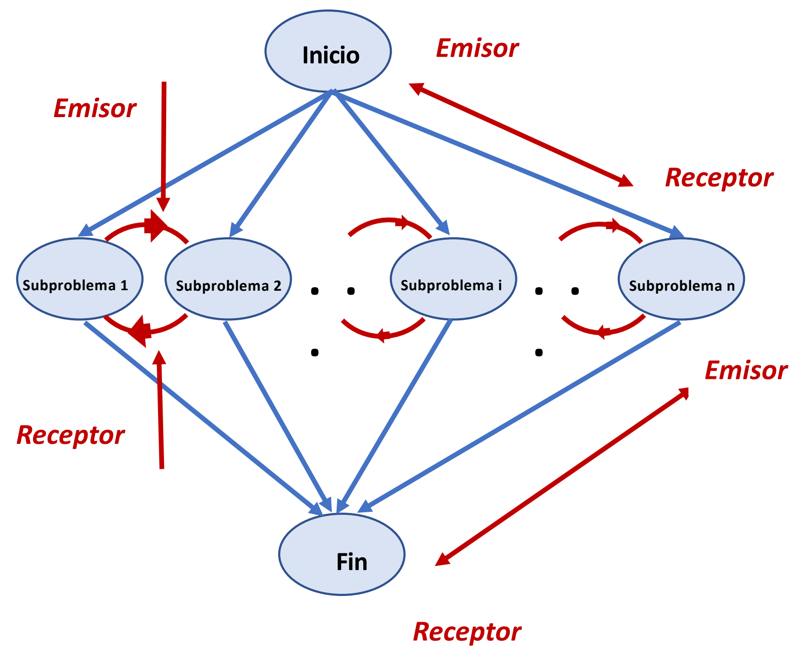
El Proceso Maestro puede asignar (incluso crear) los Procesos Esclavos de forma dinámica

Comunicación por Paso de Mensajes

La comunicación entre los procesos se hace por **Paso de Mensajes**.

- Un conjunto de procesos
- Una máquina de memorias locales
- Los procesos intercambian información mediante mensajes
- El ***Emisor*** y el ***Receptor*** del mensaje tienen que colaborar para que el intercambio se efectúe correctamente
- El programador es responsable del ***envío*** y la ***recepción*** de mensajes
- El paso de mensajes se hace mediante una llamada a una librería. Es software




Inicio
Emisor
Emisor
Receptor
Subproblema 1
Subproblema 2
Subproblema i
Subproblema n
Emisor
Receptor
Fin
Receptor

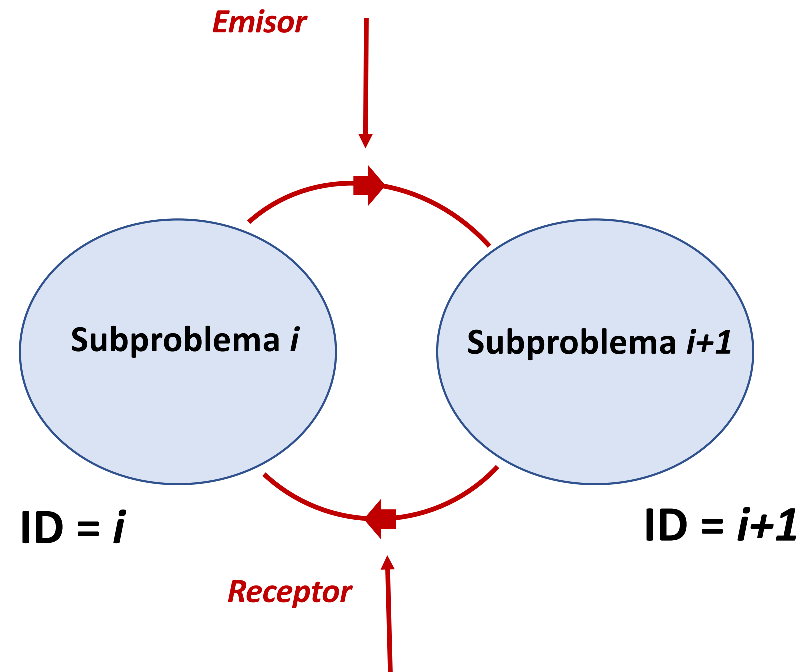
Comunicación por Paso de Mensajes

Ejemplo de código de paso de mensaje:

ID = quien_soy_yo

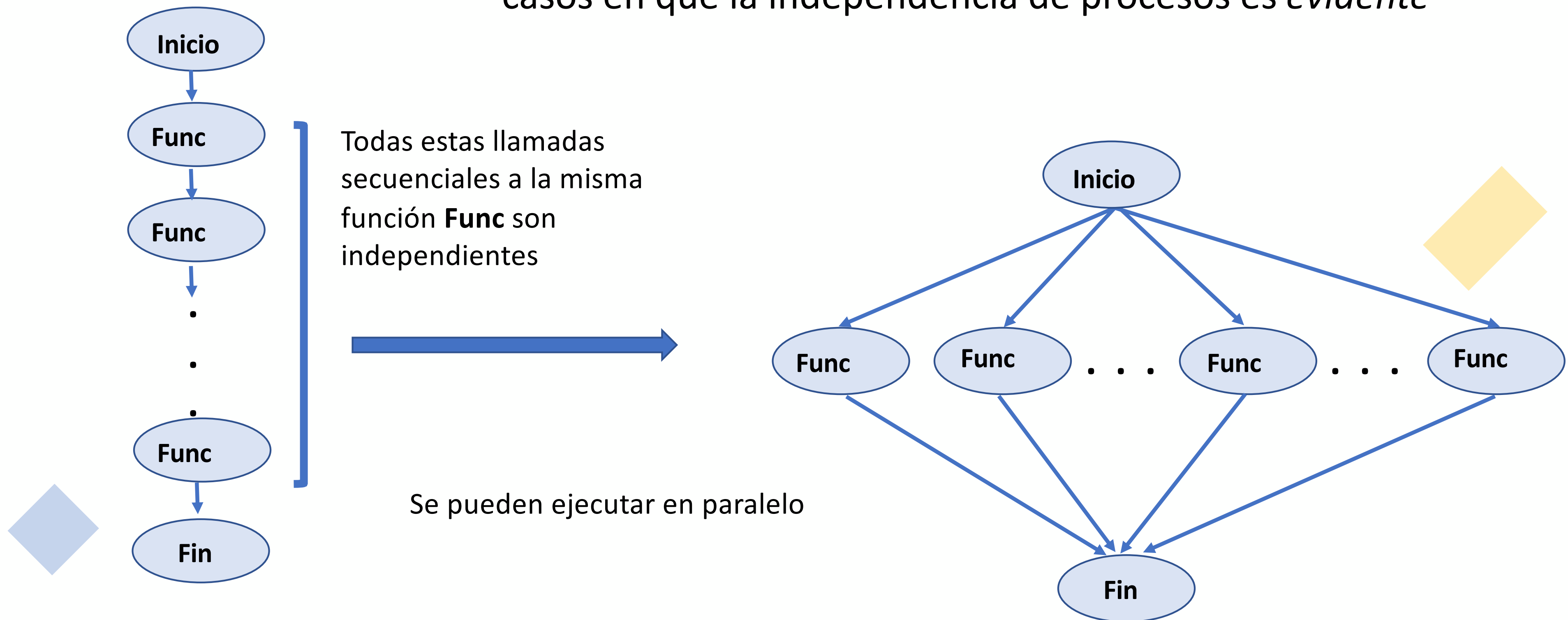
Si soy ID = ***i*** entonces
Envío Datos a ***i+1*** y espero confirmación

Si soy ID = ***i+1*** entonces
Recibo Datos de ***i*** y envío confirmación

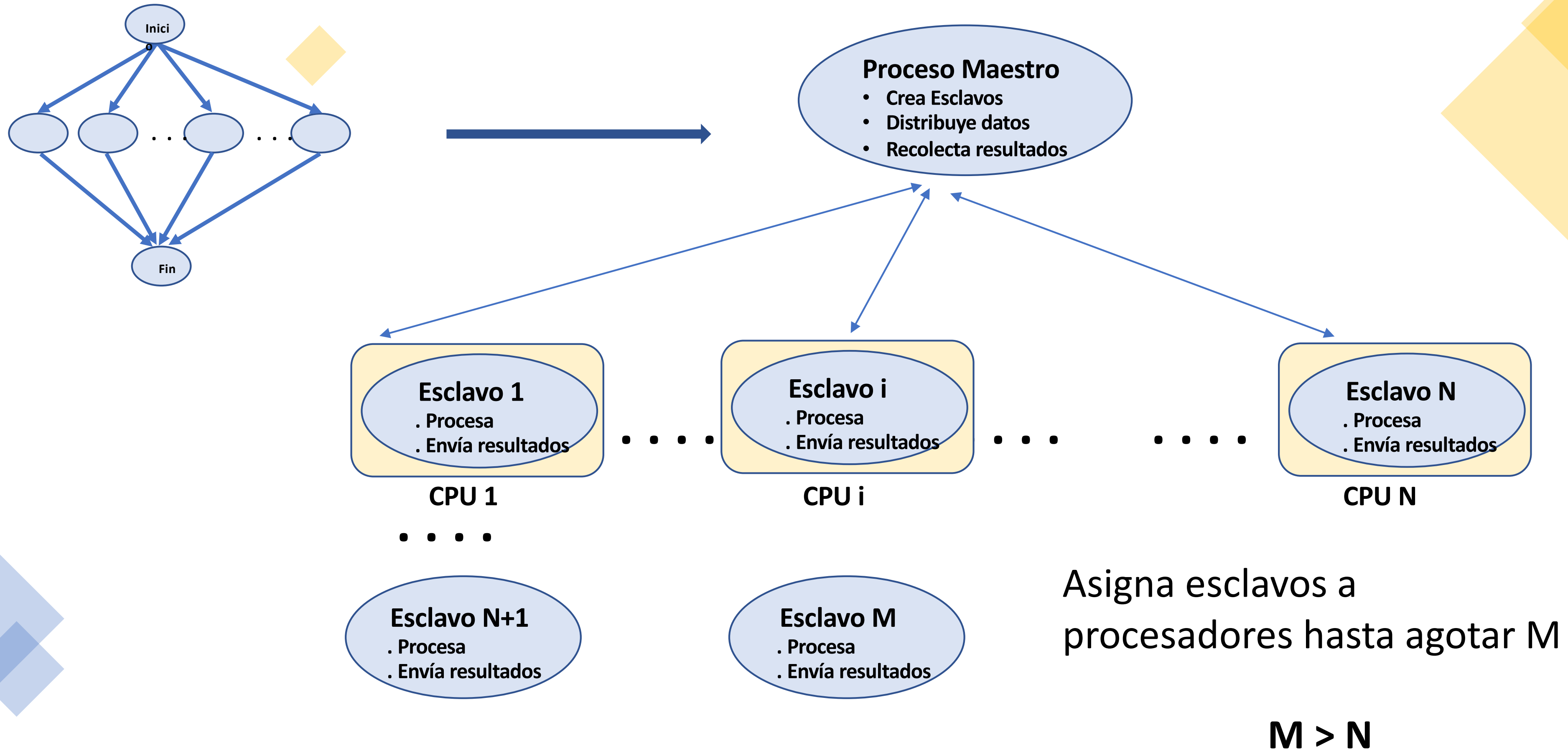


Paradigma Maestro-Esclavo

- Este paradigma es muy eficiente.
- Se usa en el caso de Paralelismo Embarazoso, ed. en los
- casos en que la independencia de procesos es *evidente*



Paradigma Maestro-Esclavo (Master-Slave)

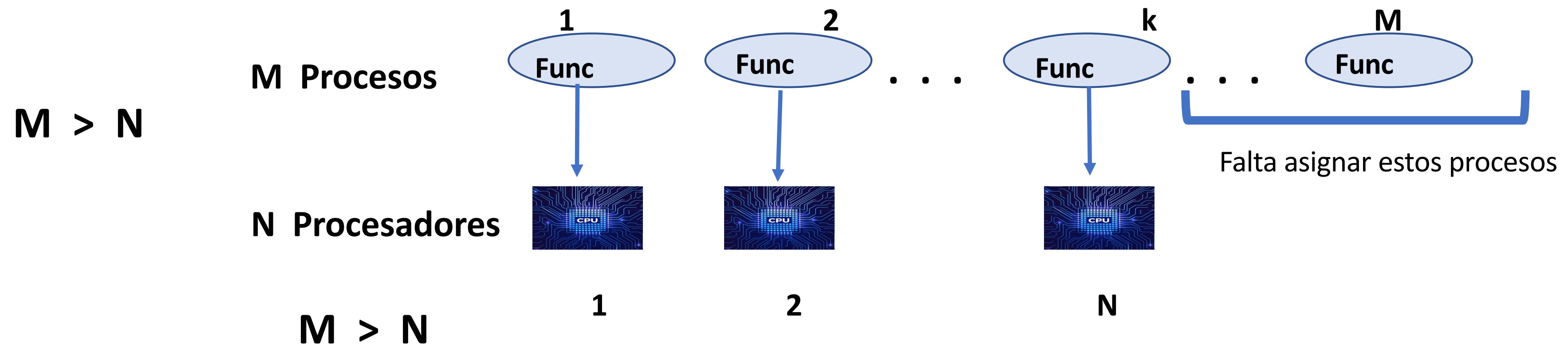


Asigna esclavos a procesadores hasta agotar M

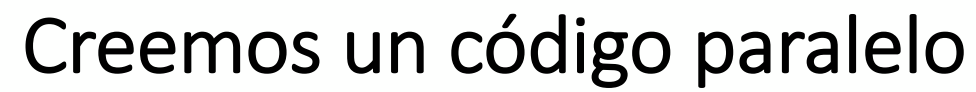
M > N

Paradigma Maestro-Esclavo

Cuando el número **N** de procesadores es menor que el número **M** de subprocesos independientes (o conjuntos de datos)



Es imposible ejecutar ***simultaneamente*** los **M** proceso en **N** procesadores


Creemos un código paralelo

Como crear un código paralelo

La librería “parallel” (“2011), nos permite ejecutar códigos paralelos. Esta librería ofrece diversas formas de paralelizar los cálculos mediante un clúster.

Pasos para crear un código paralelo

1. Crear un cluster e iniciar M procesos.
2. Enviar los datos requeridos para cada tarea a los procesos.
3. Establecer la tarea que se va a paralelizar y ejecutar en los nodos
4. Esperar a que todos los procesos terminen y obtener los resultados.
5. Cerrar los procesos. Limpiar los nodos del cluster.

https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf

https://cran.r-project.org/web/packages/parallelly/parallelly.pdf

Operaciones usando Clusters

Un “cluster” es una agrupación de objetos. En nuestro caso es una agrupación de nodos de cálculo.

En r y rstudio, explotamos paralelismo utilizando un cluster de nodos del tamaño que el programador especifique. También se puede preguntar al sistema de cuantos nodos dispone para un cluster.

Algunas funciones que nos interesan de "parallel library"

R es en su núcleo un lenguaje de programación funcional, es decir, en general

- todo lo que existe es un objeto
- todo lo que se ejecuta es una llamada a una función.

Estas son algunas de las funciones de la librería "parallel" que implementan las funcionalidades que necesitaremos.

Comencemos por **clusterApply**

clusterApply	**parApply**
clusterApplyLB	**parCapply**
clusterCall	**parLapply**
clusterEvalQ	**parRapply**
clusterExport	**parSapply**
clusterMap	**splitIndices**
clusterSplit	**stopCluster**
makeCluster	

Creando un Código Paralelo con clusterApply()

clusterApply(cluster, x, fun, ..., stopOnError = FALSE, progressBar = TRUE)

- *Cluster: el cluster donde se ejecutará la función*
- *X: número de tareas que serán enviadas a los nodos del cluster*
- *fun: la función que se ejecutará en los nodos*
- *... : parámetros adicionales para la función*
- *stopOnError = FALSE : Detener cuando uno de los hilos informa de un error? Si*
- *es FALSO, todos*
- *los errores se informarán al final.*
- *progressBar = TRUE : ¿mostrar la barra de progreso?*

clusterApply()

- Esta función de **r** aplica la función “fun”, que programemos, a los datos que especificamos en cada nodo.
- El resultado de la llamada a la función fun en cada nodo se devuelve como un elemento de la lista devuelta por clusterApply.
- Mas adelante veremos algunas variaciones de clusterApply, un poco más sofisticadas.

Veamos la creación de un código paralelo, bastante simple, paso a paso

#install.packages('parallel')

library(parallel)

#detectamos el número de nodos

detectCores()

detectCores(all.tests = FALSE, logical = TRUE) *# nodos lógicos. Devuelve la cantidad de nodos disponibles tomando en cuenta la carga actual del sistema*

detectCores(logical = FALSE) *# nodos físicos.* Devuelve la cantidad de núcleos físicos sin tomar en cuenta la carga actual del sistema.

Nota 1: el resultado depende del Sistema Operativo.

Nota 2: si solo nosotros estamos utilizando el computador y no estamos corriendo otro software, entonces usamos

detectCores()

En una MacBook Air, ambas instrucciones arrojan el mismo resultado. Yo soy el único usuario y, en principio, solo está ejecutándose el Sistema Operativo.

```
R 4.1.0 · ~/
>
>
> detectCores(all.tests = FALSE, logical = TRUE) # nodos lógicos
[1] 8
>
> detectCores(logical = FALSE) # nodos físicos
[1] 8
>
```

 # Iniciar el cluster
Cl <- makeCluster(n)

 # código a ejecutar en el cluster

 # cerramos el cluster
Stop cluster(cl)

Observaciones:

- La función “makeCluster()” crea una red de procesadores del tamaño que le indiquemos.
- Antes de crear el cluster, los nodos no tienen ningún tipo de información, no están relacionados, no hay instrucciones cargadas, las memorias cache están limpias. Nosotros debemos cargar todos los objetos necesarios.
- Cando inicializamos el cluster, estamos cargando un entorno R en los nodos, el mismo entorno para todos los nodos
- Con “stopCluster()” cerramos todos los ambientes de R creados en los nodos, si esto no se hace, nuestros procedimientos posteriores pueden presentar problemas.

Mas adelante veremos formas de asegurarnos de que el cluster quede limpio.

Observación importante: también puedo un cluster donde el número de nodos es mayor que los nodos físicos de la computadora, entonces estaré creando un cluster de nodos lógicos

Por ejemplo, en mi máquina hay 8 nodos físicos. Supongamos que creo un cluster de 12 nodos

 # Iniciar el cluster
Cl <- makeCluster(12)

En este caso los 8 procesadores físicos ejecutaran los primero 8 procesos, los 4 procesos restantes deben esperar a que se desocupen los nodos

 # Creamos la función que se ejecutará en todos los nodos

```
fun <- function(x) {
  return(x^2)
}
```

#. Ejecutamos el código con #clusterApply”

clusterApply(cluster, 1:10, fun)

Ejecutemos el código en r: CreandoCódigo 1.R

¿Cómo sabemos si este código es rápido? ¿he ahorrado tiempo de ejecución? Ese es mi propósito, entonces necesito herramientas que me permitan responder esta pregunta

- **Observación**: es posible crear y ejecutar funciones distintas en nodos distintos. Ya lo veremos más adelante

Fin de la primera parte