Declaración jurada. Declaro bajo juramento que realizaré este examen de forma estrictamente individual. En caso de que se sospeche lo contrario, soy consciente de que se me aplicará una calificación de cero y el debido proceso estipulado en el Reglamento de Orden y Disciplina de los Estudiantes de la Universidad de Costa Rica, que plantea sanciones de hasta seis años de suspensión.

Factorización prima híbrida

Desarrolle en C++ una solución distribuida y concurrente que aproveche varias máquinas y CPUs para encontrar la factorización prima de listas de números enteros ingresados en archivos. Su programa recibe por **entrada estándar** los nombres de archivos. A esta entrada se le llamará un *job*. Note que su programa no recibe el *job* por argumento de línea de comandos, sino por la entrada estándar, la cual puede o no ser redireccionada. El siguiente es un ejemplo de ejecución, donde los textos en negrita son los comandos emitidos por el usuario.

```
$ ls

fact  input001.txt  input002.txt  input003.txt  job1.txt

$ cat job1.txt

input001.txt
input002.txt
input003.txt

$ cat input001.txt

1
7
-25
87
378
1400

$ mpiexec -np 2 -f ~/hosts-mpich ./fact < job1.txt

input001.txt 7 0.001189s
input001.txt 4 0.000318s
input002.txt 15 0.000993s

$ ls

fact           job1.txt
input001.txt   input002.txt   input003.txt
output001.txt  output002.txt  output003.txt

$ cat output001.txt

1: NA
7: 7
-25: invalid number
87: 3 29
378: 2 3^3 7
1400: 2^3 5^2 7
```

Al invocarse su programa con un *job*, genera un reporte en la **salida estándar** que tiene una línea por cada archivo procesado y *en el mismo orden*, que incluye: el nombre del archivo de entrada, la cantidad de números en él,, y el tiempo en segundos que su programa tardó en procesar el archivo desde que el archivo fue abierto hasta que se completó la factorización de todos sus números.

Además del reporte, su programa debe generar por cada archivo de entrada un archivo de salida que contiene el resultado de la factorización prima de cada número, como `output001.txt` en el ejemplo anterior. Por sencillez usted puede escoger el esquema del nombre de los archivos de salida a conveniencia, siempre y cuando se pueda saber a cuál archivo de entrada pertenecen, por ejemplo `input001.out` o `input001.txt.out`.

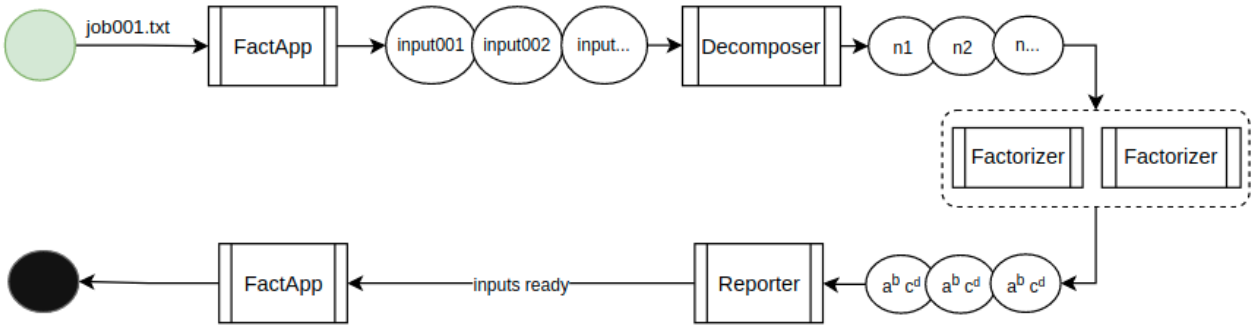
1. Diseño [10%]

Diseñe con una cadena de producción y UML (no pseudocódigo), una solución híbrida, es decir, distribuida y concurrente que aproveche lo más equitativamente posible varias máquinas disponibles en un clúster de computadoras y en ellas los núcleos de CPU, de acuerdo a los siguientes lineamientos:

1. [2%] Un proceso, representado por un objeto controlador, se encarga de leer la lista de archivos de números y distribuirla a los demás procesos usando un **mapeo dinámico**.
2. [2%] Los demás procesos reciben en hilos productores el trabajo de factorización asignado y lo descomponen en unidades más finas, para balancear la carga de trabajo entre los núcleos de CPU. Los hilos productores encolan las unidades finas resultantes en la cadena de producción. Si las unidades finas de trabajo son válidas, las redirigen a los factorizadores (punto 3), sino al consumidor (punto 4).
3. [2%] Los hilos ensambladores de paralelismo de datos, uno por cada núcleo de CPU disponible en la máquina, se encargan de factorizar las unidades finas de trabajo. El mapeo de las finas unidades de trabajo a los hilos factorizadores debe ser **dinámico**.
4. [2%] Un hilo consumidor determina si el trabajo asignado al proceso ha sido completado. Además acumula estadísticas que serán reportadas en el informe final en la salida estándar.
5. [2%] Los procesos secundarios escriben los resultados en archivo y envían el resumen al proceso que distribuye el trabajo. Cuando este último termina de recibir todos los resultados de un trabajo (*job*), los imprime en el mismo orden en que fueron ingresados en la entrada estándar.

Su diseño debe hacer claro cuáles clases son hilos y cuáles no en la cadena de producción de acuerdo al patrón **productor-consumidor**. El diseño también debe indicar el rol de cada hilo que forme parte de la cadena de producción. Además debe indicar claramente cuando los objetos pertenecen a distintos procesos. El diseño debe describir con UML las estructuras de datos usadas en cada etapa de la cadena de producción. Es decir, para cada cola debe indicarse el tipo de datos almacenado en ellas (nombre, atributos, y métodos).

Puede tomar el siguiente diseño inicial como punto de partida. En este diseño los rectángulos representan objetos e hilos de ejecución, las elipses representan nodos en colas, y los hilos dentro de un rectángulo punteado indican un equipo.



2. Solución distribuida [40%]

Utilice la tecnología **MPI** para distribuir el trabajo de factorización entre varios procesos de acuerdo al diseño planteado. Rubros a evaluar:

1. [11%] Un único proceso lee de la entrada estándar el nombre de los archivos y distribuye el trabajo entre los demás procesos mediante un mapeo dinámico. Su programa debe poder funcionar con la cantidad de procesos que el usuario desee crear (con **`mpiexec`**). Si el usuario sólo crea un proceso, éste realizará todo el trabajo. Puede suponer que todos los archivos se encuentran en la misma carpeta donde es invocado su programa.
2. [8%] Cada proceso secundario recibe el trabajo asignado y lo pone en la cadena de producción.
3. [11%] Una vez que el trabajo ha sido procesado en la cadena de producción, se envían al proceso repartidor los estadísticos para el informe. Los pasos anteriores se repiten hasta que no haya más trabajo pendiente.
4. [10%] El proceso repartidor recibe los estadísticos, los ordena, e imprime el reporte en el mismo orden en que ingresaron los nombres de archivos en la entrada estándar.

Los puntos 1 y 4 deben ser realizados por una clase controladora que puede correr en el hilo principal del proceso. El código debe estar modularizado y no en extensas subrutinas.

3. Solución concurrente [40%]

Implemente una solución concurrente del diseño planteado usando el **patrón productor-consumidor** visto en clases. Está prohibido usar Pthreads u OpenMP. Rubros a evaluar:

1. [15%] Un hilo secundario productor recibe trabajo del proceso que reparte. Lee del archivo el trabajo asignado, lo carga en estructuras de datos, y lo descompone en unidades finas. El trabajo válido es colocado en la cola para factorización. El trabajo inválido es enviado a la cola de trabajo finalizado de tal forma que no pase por los factorizadores.
2. [10%] Los hilos ensambladores de paralelismo de datos, uno por cada núcleo de CPU disponible en la máquina, consumen las unidades de trabajo pendientes y se encargan de factorizar los números. Colocan los resultados en la cola de trabajo finalizado.
3. [15%] Un hilo secundario consumidor se encarga de determinar si todas las unidades finas del trabajo asignado han sido procesadas. En tal caso, escribe los resultados de la factorización en el archivo de salida en el mismo orden en que fueron leídos del archivo. Además genera datos estadísticos para el reporte y los envía al proceso que reparte con el fin de que los presente al usuario.

4. Análisis de resultados [10%]

Compare el desempeño de los tres modos de ejecución siguientes de su solución en el clúster de Arenal.

1. *Serial*: en un nodo esclavo del cluster con un solo hilo.
2. *Concurrente*: en un nodo esclavo del cluster con tantos hilos como CPUs tenga el nodo.
3. *Distribuida*: en el cluster arenal utilizando todos los nodos esclavos disponibles y tantos hilos como hayan CPUs disponibles en los nodos esclavos. Puede utilizar el nodo máster para el proceso que lee la entrada estándar.

Sus productos serán una hoja de cálculo, un gráfico comparativo, y una discusión.

1. [2.5%] Realice todas las mediciones en el cluster arenal utilizando el caso de prueba **`job2.txt`**. Utilice la configuración correcta de las mediciones en cada una de las versiones del programa (número de procesos, hilos, nodos). Anote sus resultados en una hoja de cálculo. Puede tomar como base para la captura de tiempos el archivo *captura_de_tiempos.ods* que se le entregó.

2. [5%] Cree un gráfico comparativo. En el eje X muestre la versión del programa (serial, concurrente, distribuido). El gráfico debe ser combinado con dos ejes Y. El eje Y primario, ubicado a la izquierda del gráfico, indica el incremento de velocidad (*speedup*). En el eje Y secundario, a la derecha del gráfico, indique la eficiencia de cada modo de ejecución. Incluya las dos series en el gráfico, la de incremento de velocidad y de la eficiencia, acorde a su eje.
3. [2.5%] Cree un reporte breve con una discusión de máximo 500 palabras, en la que compare las versiones de su solución basado en el gráfico y razone sobre qué produjo el mayor incremento de desempeño.

Entrega de la solución

Entregue su solución en su repositorio de control de versiones personal para el curso, en una subcarpeta `ampliacion/`. Se recomienda realizar *commits* y *push* con frecuencia y no sólo al final de la prueba. Puede disponer de otros archivos realizados durante el curso, sean de ejemplos hechos en clase por los docentes o código suyo de tareas o de proyectos. De usar código de cualquier otra fuente debe dar el debido crédito como se estipula en la normativa universitaria.

En cada rubro se evalúan aspectos y buenas prácticas de programación como las siguientes. Que el código fuente compile (puede comentar/deshabilitar código en edición que no lo haga en el *commit* final). Que sea correcto (generar los resultados esperados). La calidad del código fuente, por ejemplo, modularizar (dividir) subrutinas o clases largas, y modularizar los archivos fuente (varios .hpp y .cpp). Apego a una convención de estilos (cpplint). Uso identificadores significativos. Inicialización de todas las variables. Reutilización de código. No generar condiciones de carrera, espera activa, bloqueos mutuos, inanición, accesos inválidos ni fugas de memoria. Su solución debe tener mecanismos para compilar el programa.

Importante. La solución será calificada con cero si no realiza la descomposición y mapeos como se indica en el enunciado. Debe considerar las implicaciones que la distribución, descomposición y mapeo tienen en las estructuras de datos para mantener la trazabilidad de los mismos.