Universidad de Costa Rica

Facultad de Ingeniería
Escuela de Ciencias de la Computación e Informática

CI0117 – PROGRAMACIÓN PARALELA Y CONCURRENTE
Profesor Jeisson Hidalgo

Tarea Programada 1

Estudiante:
Emmanuel D. Solís - B97670
emmanuel.solispomares@ucr.ac.cr

20 de julio de 2021

Índice

I - S - 2021 - RRF - Programación Paralela y Concurrente - 002

Tarea03: goldbach_optimization

Realice **al menos dos** optimizaciones de su solución para el cálculo de sumas de Goldbach que incrementen el desempeño respecto a la versión anterior (`goldbach_pthread`). Es requerido proveer evidencia de este incremento con datos, gráficas, y su respectiva discusión textual, en un documento de reporte. Además compare el comportamiento de su solución ante diferentes niveles de concurrencia.

En todas las optimizaciones es obligatorio seguir el método sugerido para optimizar visto durante las lecciones del curso. Este método es cíclico, y se deberá recorrer al menos dos veces que logren incrementar el desempeño de la solución respecto a la versión previa.

Importante: Si recibió observaciones sobre la Tarea02 aplíquelas antes de continuar, como se indica en la sección “Correcciones a la Tarea02” al final del enunciado.

Para su entrega copie su solución concurrente `goldbach_pthread` a la carpeta `tareas/goldbach_optimization` en su repositorio de control de versiones. Cree un archivo homónimo a la carpeta en formato de `Markdown`, `AsciiDoc`, o LaTeX al cual se le referirá como **documento de reporte**. Este documento tendrá una sección por cada iteración del método sugerido para optimizar. En la sección documente el diseño de la modificación que piensa incrementará el desempeño, el rendimiento (speedup y eficiencia) del código antes y después de modificarlo, y una discusión de las lecciones aprendidas.

¿Cómo medir el tiempo de ejecución?

Para todas las mediciones de rendimiento use el caso de prueba `input023.txt` en un nodo esclavo (y no la máquina máster) del clúster Arenal. Anote los resultados en una hoja de cálculo que le facilite realizar las comparaciones. Se adjunta una hoja de cálculo modelo que puede usar para este fin. Para poder medir el rendimiento puede (1) modificar su programa, o (2) usar una herramienta como `perf`.

1. Si modifica su programa, registre el tiempo de *CPU wall time* con precisión de nanosegundos que tarda en ejecutar cada uno de los casos de prueba que utiliza para las mediciones. Utilice para ello la subrutina `clock_gettime` que se demostró durante las lecciones del curso. Para que el reporte de la duración no interfiera con la salida estándar (que se compara con los casos de prueba), haga que el programa imprima la duración en el error estándar.

2. Si prefiere usar `perf` instale el paquete correspondiente (`linux-perf` en Debian o `perf` en RedHat). Si en Debian requiere permisos de administración para permitir a `perf` inspeccionar a otros programas, puede requerir configuración adicional para correrlo como usuario normal (como crear el archivo `sudo echo 1 >/proc/sys/kernel/perf_event_paranoid` y reiniciar). Una vez instalado, puede obtener la duración con `perf stat myprogram < tests/input###.txt`.

Para toda medición deben realizarse tres corridas y tomar la menor de ellas. Algunas mediciones dependen de la cantidad de CPUs disponbies en la máquina, la cual la puede consultar con el comando `lscpu` de Linux. Todas las ejecuciones deben realizarse en la misma máquina: un nodo esclavo del

clúster del arenal (no la máquina máster), identificado con el número `N` entre 0 y 3 en los siguientes comandos como guía:

```
ssh CARNET@arenal.ecci.ucr.ac.cr
git clone url_mi_repositorio                # Solo la primera vez
cd mi_repositorio/tareas/programa_a_medir
make

ssh compute-0-N
cd mi_repositorio/tareas/programa_a_medir
./my_program num_threads < tests/inputXYZ.txt   # Anotar duración
./my_program num_threads < tests/inputXYZ.txt   # Anotar duración
./my_program num_threads < tests/inputXYZ.txt   # Anotar duración
# Tomar la menor de las tres duraciones anteriores
```

Optimización01: Mapeo dinámico [30%]

La primera optimización es implementar mapeo dinámico de los números. Es decir, los hilos secundarios se reparten los números que se leyeron de la entrada estándar de forma dinámica. Su implementación del mapeo dinámico debe seguir el patrón productor-consumidor, donde el productor puede ser el hilo principal, y los consumidores son los hilos secundarios. La cantidad de hilos secundarios debe poderse ajustar mediante un argumento de línea de comandos. Si éste se omite, se debe asumir la cantidad de núcleos de procesador disponibles en el sistema.

De acuerdo al patrón, la comunicación entre los productores y consumidores se realiza mediante un *buffer* acotado o no, que puede ser una cola *thread-*

safe. Sin embargo, si implementa además seguridad condicional (un arreglo de resultados), puede simplificar el *buffer*, por ejemplo, mediante un contador entero protegido por *mutex*.

Siga todos los pasos del método sugerido para optimizar. Como paso 1, haga una medición en un nodo esclavo del clúster Arenal de la tarea02 con ocho núcleos tomando la menor duración de tres corridas con el caso de prueba `023`. Nota: Para esta optimización **no** es necesario realizar el paso 2 de *profiling*. El paso 3 corresponde a este mapeo dinámico. Las mediciones en el paso 5 debe realizarlas en la misma máquina, con el mismo caso de prueba, y siempre tomando la menor de tres corridas.

[**Opcional 10% extra**] Haga más granular la unidad de descomposición. Mapear dinámicamente los *números* ingresados en la entrada estándar puede provocar desequilibrio. Por ejemplo, un número muy grande entre varios pequeños, provocaría que el hilo que se encarga del número grande tarde mucho mientras los otros hilos queden ociosos. A modo de sugerencia puede repartir sumas o grupos de sumas de cada número ingresado por el usuario en la entrada estándar.

Optimización02: Propuesta libre [30%]

La segunda optimización es propuesta libremente por el o la estudiante, siempre y cuando demuestre empíricamente que incrementa el desempeño. Realice todos los pasos del método sugerido para optimizar. En el paso 1 tome como punto de partida los resultados de la optimización 1.

En el paso 2 realice un análisis dinámico de rendimiento (*profiling*) para asegurarse de que su propuesta de optimización impacta directa o indirectamente las líneas de código más críticas en consumo de CPU. Para realizar el análisis dinámico de código, ejecute su solución actual con la

herramienta `callgrind` con un caso de prueba liviano (por ejemplo `input008.txt`). Visualice la salida con `KCachegrind` y determine las regiones de código que más demandan procesamiento o aquellas que las invocan. Haga una captura de pantalla y agréguela a su documento de reporte y explique resumidamente cómo el diseño de su optimización piensa disminuir este consumo.

En el paso 3 y 4 documente rápidamente (un párrafo o menos) en qué consiste la optimización en su documento de reporte, luego implemente la potencial mejora. En el paso 5 recuerde siempre hacer tres corridas en las mismas condiciones que la optimización 1, es decir: en la misma máquina, con el mismo caso de prueba, con ocho hilos secundarios, y tomar la menor duración de tres corridas. Anote los resultados en su hoja de cálculo y estime el incremento de velocidad. Escriba en su documento de reporte estos valores. Si no logró incrementar el rendimiento, conjeture en su discusión a qué se podría deber ese resultado.

Comparación01: de optimizaciones [20%]

Ya que ha realizado varias optimizaciones, compárelas entre ellas para determinar cuál aportó más al incremento del desempeño o genera mejor eficiencia. Todas las mediciones han de ser comparables, por tanto, deben realizarse en la misma máquina (nodo esclavo del clúster Arenal), con el mismo caso de prueba (`023`), y tomar la menor duración de tres corridas. Las versiones de sus programas de Goldbach a medir son:

Versión	Descripción
`tarea01`	La versión serial
`tarea02`	La versión concurrente
`optim01`	La versión con mapeo dinámico

`optim02`	La versión con su segunda optimización

Nota: A la tabla anterior puede agregar la `optim01b` si realizó la opcional de descomponer a nivel de sumas. Asimismo, `optim03`, `optim04`, y subsiguientes, si realizó varias iteraciones por el método sugerido para optimizar que lograran incrementar el desempeño.

Todas las mediciones deben realizarse con tantos hilos como la cantidad de núcleos disponibles en el sistema (ocho para el nodo esclavo del clúster Arenal), a excepción de la versión serial. Si no tiene alguna de las mediciones anteriores realícela y agréguela a la hoja de cálculo. Para todas las mediciones calcule el incremento de velocidad respecto a la versión serial, y la eficiencia.

Cree un gráfico en la hoja de cálculo. En el eje X muestre la versión del programa, correspondiente a la columna "Versión" de la tabla (`tarea01`, `tarea02`, `optim01`, `optim02`). El gráfico debe ser combinado con dos ejes Y. El eje Y primario, ubicado a la izquierda del gráfico, indica la duración de las mediciones en segundos. Este debe ser el eje de referencia para serie de mediciones de tiempo. El eje Y secundario, a la derecha del gráfico, indica el incremento de velocidad (*speedup*) y aunque no tiene unidades puede indicar la palabra "veces". Este debe ser el eje de referencia para la serie de incrementos de velocidad.

Incluya una sección "Comparación de optimizaciones" su documento de reporte. Haga una discusión de máximo 500 palabras, en la que compare las versiones de su solución e indique qué produjo el mayor incremento de desempeño. Exporte el grafo a formato SVG o PNG e incrústelo en la discusión.

Comparación02: grado de concurrencia [20%]

Ahora que dispone de programas medibles, uno de los aprendizajes más enriquecedores es poder compararlos en diversas circunstancias, como diferente hardware, casos de prueba, y grado de concurrencia, entre otros. En esta sección podrá comparar sus soluciones respecto a este último rubro.

Todas las mediciones han de ser comparables, por tanto, deben realizarse en la misma máquina (nodo esclavo del clúster Arenal), con el mismo caso de prueba (`023`), y tomar la menor duración de tres corridas. Lo que varía en cada medición es la cantidad de hilos de ejecución, obtenidas en función de la constante `c` definida como la cantidad de núcleos de procesador disponibles en un sistema. Por ejemplo, para un nodo esclavo del clúster Arenal, `c` equivale a 8 núcleos.

Para realizar las comparaciones, se usarán 7 niveles de concurrencia en función de `c`, es decir, usted registrará en su hoja de cálculo 7 mediciones de la duración de sus programas, listadas en la tabla a continuación. Dado que para lograr una medición más fiable, debe ejecutar su programa al menos tres veces y tomar la duración de menor duración, en total realizará al menos 21 ejecuciones de sus programas. En la última columna de la siguiente tabla se muestra la cantidad de hilos para el nodo esclavo del clúster Arenal.

#	Leyenda	Descripción	Hilos
1	S	Serial	1
2	1	Un solo hilo	1
3	.5C	Tantos hilos como la mitad de CPUs hay en la computadora que ejecuta el programa	4
4	1C	Tantos hilos como CPUs hay en la computadora que ejecuta el programa	8
5	2C	Dos hilos por cada CPU que hay en la computadora que ejecuta el programa	16

6	4C	Cuatro hilos por cada CPU que hay en la computadora que ejecuta el programa	32
7	D	Tantos hilos como números recibe el programa en la entrada estándar	D

La medición 1 en la tabla (`S`) corresponde a la menor de tres (o más) ejecuciones de su programa serial (`goldbach_serial`) con el caso de prueba `023`. Las restantes seis mediciones deben realizarse con el programa que obtuvo tras la Optimización02 con tres ejecuciones del caso de prueba `023`. Lo que varía en cada medición es la cantidad de hilos a crear, el cual es el argumento con que se invoca su programa en línea de comandos. Anote la menor de las tres (o más) ejecuciones en su hoja de cálculo. Note que la Optimización02 ya obtuvo la medición para `1C`.

Una vez que tenga todas las mediciones calcule el incremento de velocidad (*speedup*) de cada medición respecto a la serial (incluso la serial misma). Calcule también la eficiencia de cada medición.

Cree un gráfico en la hoja de cálculo. En el eje X muestre la cantidad de hilos que realizaron los cálculos durante la ejecución del programa, correspondiente a la columna “Leyenda” de la tabla (S, 1, .5C, 1C, ...). 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*) y aunque no tiene unidades puede indicar la palabra “veces”. Este debe ser el eje de referencia para la serie de incrementos de velocidad. El eje Y secundario, a la derecha del gráfico, indica la eficiencia obtenida por cada nivel de concurrencia. De esta forma permitirá comparar el incremento de desempeño (*speedup*) contra la eficiencia.

Finalmente, realice un análisis de los resultados obtenidos en la parte final de su documento de reporte. Exporte los gráficos a formato SVG o PNG e

incrústelos como imágenes. Esta discusión debe ser concisa, de máximo 500 palabras, con su interpretación de los gráficos. A partir de sus resultados responda a la pregunta ¿cuál es la cantidad de hilos óptima para conseguir el mejor rendimiento?. Considere tanto el incremento de velocidad como la eficiencia en su respuesta.

Evaluación

Los pesos de cada rubro se encuentran en los títulos de las secciones anteriores. La siguiente es una lista de chequeo de los productos a entregar en su repositorio serán:

1. El código fuente de la Optimización01 y Optimización02.
2. El reporte en formato Markdown, AsciiDoc, o LaTeX, con una sección por cada iteración del método sugerido para optimizar, sus resultados y lecciones aprendidas.
3. La hoja de cálculo con las mediciones, incrementos de velocidad, y eficiencia.
4. Los gráficos combinados dentro de la hoja y exportados a imágenes.
5. Las discusiones de las dos comparaciones y sus gráfico incrustados en el reporte.

Respecto al código fuente se evaluarán las buenas prácticas, como:

1. La corrección de la solución a partir de los casos de prueba.

2. La completitud de la solución. Es decir, si realiza todo lo solicitado.

3. La calidad de la solución ejecutable. Por ejemplo, no generar condiciones de carrera, espera activa, bloqueos mutuos, inanición (puede usar herramientas como `helgrind`, `tsan`). Tampoco accesos inválidos ni fugas de memoria (`memcheck`, `asan`, `msan`, `ubsan`).

4. La calidad del código fuente, al aplicar las buenas prácticas de programación. Por ejemplo, modularizar (dividir) subrutinas o clases largas, y modularizar los archivos fuente (varios `.hpp`, `.cpp`, `.h`, `.c`). Apego a una convención de estilos (`cpplint`). Uso identificadores significativos. Inicialización de todas las variables. Reutilización de código

5. La documentación de interfaces de clases, métodos, y tipos de datos, con Doxygen. La salida de Doxygen debe estar en una carpeta `doc/` y ésta NO se debe agregar a control de versiones. Documentar el código no trivial en los cuerpos de las subrutinas.

Otras prácticas que serán consideradas:

1. Buen uso del repositorio de control de versiones. Cada *commit* debe tener un cambio con un significado, un mensaje significativo.

2. Debe mantenerse la estructura de archivos y directorios característica de un proyecto de Unix, como se solicitó en Tarea02.

3. El buen uso de la comunicación escrita, tal como lo estipula la carta al estudiante.

Se recomienda subir una copia comprimida del código fuente a este enunciado a modo de respaldo.

Anexo: Correcciones a la Tarea02

Si en la tarea anterior obtuvo observaciones, impleméntelas para optar por la mejora en la calificación. Corrija su solución a la `tarea02` en la carpeta `tareas/goldbach_pthread/` para aplicar las observaciones que obtuvo durante la revisión de la misma, de manera que no se repliquen las

deficiencias identificadas en la `tarea03`. Recuerde utilizar issues de su sistema de control de versiones:

1. Por cada observación que recibió durante la revisión (sea que las haya anotado o que visualice la grabación de la revisión), cree un issue en el sistema de alojamiento de control de versiones (`git.ucr` o `github`). Inicie el título del issue con el número de la tarea, por ejemplo "Tarea02: eliminar la serialización innecesaria". Describa en el issue el problema identificado puntualmente. Note que cada issue recibe un número que lo identifica.

2. Corrija un issue a la vez en su repositorio. Modifique los archivos que necesite directamente (NO cree una copia de carpeta). Para cada corrección cree un *commit* y en el mensaje refiera el número del issue. Si es el último *commit* que resuelve el issue, ciérrelo desde el mensaje de *commit* por ejemplo, el mensaje:

   ```
   git commit -m 'Store sums in dynamic array instead of a static one. Close #13'
   ```

   cerraría el issue identificado con 13. De esta forma el issue y el commit quedan ligados en el sistema de control de versiones y es fácil la rastreabilidad.

Una vez que haya corregido la `tarea02`, continúe con la `tarea03` (optimizaciones).

1. Optimizaciones

1.1. Mapeo Dinámico

1.1.1. Medición de rendimiento inicial

Para efectos de esta optimización es importante mencionar que parte de cómo estaba implementada la Tarea 2 era que ya contaba un mapeo dinámico por lo que no es posible realizar una comparación previa y una posterior por cuanto ambas son las mismas. El tiempo de ejecución que tuvo dicha ejecución para el tiempo de caso de prueba *tests_small/input*006*.txt* fue:

```
$ execution time: 1.161185000s
```

el por qué de haber usado esta prueba es solamente porque el tiempo con el que contaba antes de la fecha límite de la entrega de esta tarea no era suficiente para correr tres 3 veces las pruebas de casos largas por lo que decidí usar el caso de prueba más difícil de los pequeños para obtener un resultado similar.

1.1.2. Regiones críticas

Evidentemente el punto crítico más notorio comparado con la solución serial era la creación de las soluciones.

1.1.3. Modificaciones

En este caso lo que se agregó fue el respectivo mapeo dinámico.

1.1.4. Verificación de modificaciones

La medición sería la misma que la anterior.

```
$ execution time: 1.161185000s
```

1.1.5. Medición luego de modificación

Para la misma prueba *tests_small/input*006*.txt* el tiempo de duración seria fue de:

```
$ execution time: 5.196784000s
```

por lo que aún en una prueba pequeña se puede apreciar la mejoría que tendría un código concurrente contra uno serial.

1.1.6. Lecciones

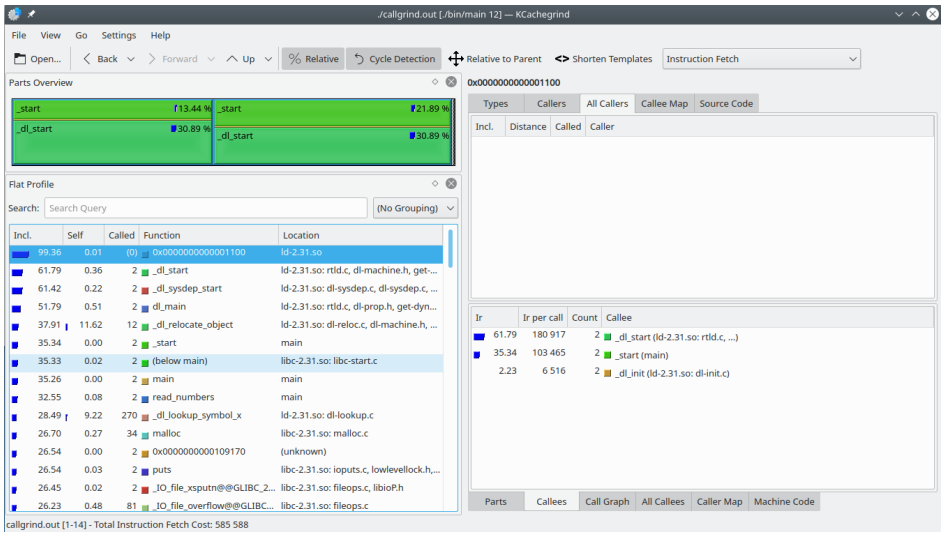
La lección más importante es que el mapeo dinámico es muy útil si no existe un gran desbalance y si se implementa correctamente porque de lo contrario puede serializar el código.

1.2. Optimización escogencia personal

1.2.1. Medición de rendimiento inicial

La medición del rendimiento previo al cambio es la misma que la del mapeo dinámico puesta además de este ningún otro ha sido efectuado, el cuál sería:

```
$ execution time: 1.161185000s
```



1.2.2. Regiones críticas

Por medio de la figura 2 pudimos observar que una de las regiones críticas son en solucionesImpares por el hechode tener que hacer el ciclo para números que se sabe no serán primos como los múltiplos de dos.

1.2.3. Modificaciones

Por lo tanto por medio de un for especial se ignoraron todos los numeros pares, disminuyendo en gran parte la cantidad de iteraciones.

1.2.4. Verificación de modificaciones

Haciendo las pruebas respectivas los resultados fueron satisfactorios por lo que es una solución valida.

1.2.5. Medición luego de modificación

Al haber concluido en pequeña escala se pudo ver un cambio del tiempo de duración hacia:

```
$ execution time: 1.088627000s
```

2. Comparación de optimizaciones

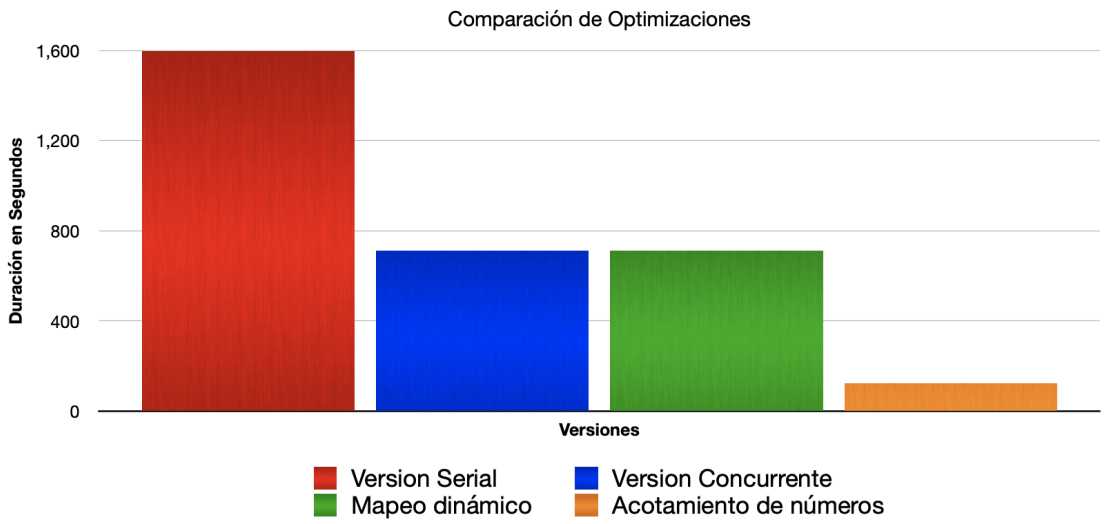
Las siguientes comparaciones se realizan para el caso de prueba mediano *input023.txt*. Los resultados se pueden ver expresados en el gráfico 2.

2.1. Comparación 1: Diferentes versiones

2.1.1. Versión serial

Se obtuvo una duración de:

```
$ execution time: 1600.00 seg.
```



2.1.2. Versión concurrente

```
$ execution time: 712.00 seg.
```

2.1.3. Versión con mapeo dinámico

```
$ execution time: 712.00 seg.
```

2.1.4. Version con optimizacion de escogencia

```
$ execution time: 125.00 seg.
```

2.2. Comparación 2: Grados de Concurrencia

Esta es una comparación para ver cuál es la cantidad óptima de hilos de ejecución, para ello podemos basarnos en el gráfico 2.2.

2.2.1. Serial

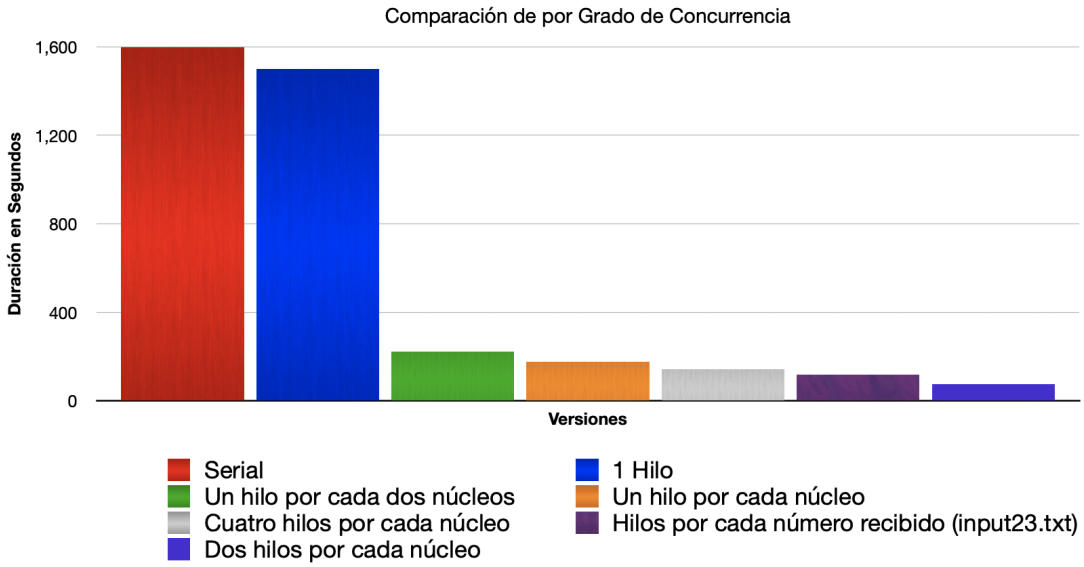
Se obtuvo una duración de:

```
$ execution time: 1600.00 seg.
```

2.3. 1: Un solo hilo

Se obtuvo una duración de:

```
$ execution time: 1500.00 seg.
```



2.4. .5C: Cantidad de hilos por mitad de disponibles por hardware

Mi computadora local tiene 8 nucleos por lo que con 4 hilos se obtuvo una duración de:

```
$ execution time: 224.00 seg.
```

2.5. 1: Cantidad de hilos a igual a diponibles por hardware

Mi computadora local tiene 8 nucleos por lo que con los 8 hilos se obtuvo una duración de:

```
$ execution time: 178.00 seg.
```

2.6. 2C: Cantidad de hilos al doble de los disponibles por hardware

Mi computadora local tiene 8 nucleos por lo que con los 16 hilos se obtuvo una duración de:

```
$ execution time: 142.00 seg.
```

2.7. 4C: Cantidad de hilos al cuádruple de los disponibles por hardware

Mi computadora local tiene 8 nucleos por lo que con los 32 hilos se obtuvo una duración de:

```
$ execution time: 118.00 seg.
```

2.8. D: Cantidad de hilos a la cantidad de números que se reciben

Para la prueba *test-medium/input023.txt* que tenia 54 números se tendría un resultado de:

```
$ execution time: 7500.seg.
```

3. Conclusiones

Por medio de las comparaciones de rendiciones en la sección anterior es que podemos llegar a la siguiente conclusión, en todos los casos que se hicieron optimizaciones se obtuvo un mejor resulto, los cambios se dieron en saltos igual de grandes. La versión paralela aumento la velocidad en mas de un 100 % y sucedio lo mismo realizando el acotamiento de números; esto sucede porque ambas optimizaciones representan un cambio significativo en el código. Sin embargo va a llegar un punto en que ya habremos optimizado la máxima cantidad posible según la *Ley de Amdahl* que dice que un programa está acotado por su parte serial.

Respecto a los aumentos de velocidad por la cantidad de hilos usados tambien se ve un aumento de velocidad, es sumamemte distante entre las versión serial y con un hilo en contra de las versiones que usan mas hilos. Entre más hilos se estén usando los tiempos se mejoran sin embargo esto en cierto punto va a dejar de suceder y al contrario va a suceder un efecto inverso pues entre más hilos hayan va a tener el sistema operativo que hacer mayor coordinación para los hilos y puede que esto sea innecesario por ejemplo si tenemos más hilos que la cantidad de números a procesar porque no tiene sentido estar haciendo ese desperdicio de recursos.