COMPILACIÓN, EJECUCIÓN Y OPTIMIZACIÓN DE PROGRAMAS

EJERCICIOS DE LABORATORIO

Introducción

Los siguientes ejercicios forman las prácticas guiadas del curso COMPILACIÓN, EJECUCIÓN Y OPTIMIZACIÓN DE PROGRAMAS.

No se proporcionará ningún documento con las soluciones a estos ejercicios. Asegúrese de que los comprende y sabe como resolverlos, y no dude en consultar al profesor en caso de duda.

Instalación de los ejercicios

Los ejercicios de laboratorio están localizados en el /tmp de la máquina del profesor, en un fichero empaquetado llamado optlabs.tar. Transfiéralo a su cuenta y su directorio local y desempaquete el fichero. Se creará un directorio llamado optlabs en su directorio local. Dentro encontrará los ejercicios clasificados en carpetas, según el tema al que hagan referencia (LAB1,LAB2,...). También encontrará un directorio llamado CLASS que contiene varios ficheros ejemplo que se usarán durante las explicaciones de pizarra.

LAB 1. Compilación y linkado

1. Compilación usando optimización automática

- Compile el código ejercicio1.f probando los diferentes niveles de optimización. Para todos ellos ejecute el código resultante y compare los tiempos de ejecución. Para medir el tiempo de ejecución puede usar el comando time.

```
time  ejercicio1
```

- ¿Qué observa?
- Compare también el tamaño de los ficheros ejecutables. ¿Qué se observa?

Optimización	**Tiempo compilación**	**Tamaño ejecutable**	**Tiempo ejecución**	**resultado**
+O0				
+O1				
+O2				
+O3				
+O4				
-O				

2. Practicar el uso ld para enlazar código previamente compilado

- Edite el fichero ejercicio2b.c. Verá que este programa usa una subrutina que se encuentra en el fichero ejercicio2a.c. Compile los ficheros ejercicio2a.c y ejercico2b.c para obtener los ficheros objeto ejercicio2a.o y ejercicio2b.o. Use ld para enlazar ambos objetos en un ejecutable. Ejecute y mida tiempos.

3. Programación multilenguaje.

- Eche un vistazo a los ficheros ejercicio3a.f y ejercicio3b.c. Ambos poseen un bloque main, que llama a una subrutina. El main de ejercicio3b.c invoca a la subrutina sub1 que se encuentra en ejercicio3c.f y el main de ejercicio3a.f invoca a la subrutina sub2 que se encuentra en ejercicio3d.c. Trataremos de compilar enlazando ambos programas.

- Llamada a C desde Fortran:
  - Para invocar a una subrutina en C desde Fortran hemos de nombrar la subrutina en C con "_". Observe que en ejercicio3d.c la subrutina se llama sub2_ y sin embargo en el main de ejercicio3a.f se invoca como sub2.
  - Compile con el compilador de C el código en C para generar un fichero objeto (ejercicio3d.o)
  - Compile con el compilador de f77 el código en Fortran (ejercicio3a.f) y el fichero objeto anterior
  - Ejecute y mida tiempos. Observe el resultado.

- Llamada a Fortran desde C:
    - Para invocar a una subrutina en Fortran desde C hemos de invocar a la subrutina en Fortran con un "_". Observe que en ejercicio3b.c invocamos a sub1_() y sin embargo en ejercicio3c.f la subrutina se llama sub1().
    - Compile con el compilador de f77 el código en Fortran para generar un fichero objeto (ejercicio3c.o)
    - Compile con el compilador de C el código en C (ejercicio3b.o) y el fichero objeto anterior
    - Ejecute y mida tiempos. Observe el resultado

- Problema con la Entrada/Salida:
    - Incluya una llamada a E/S en cada una de las subrutinas de C y Fortran anteriores (sub1 y sub2).
    - Repita los apartados anteriores con estos nuevos ficheros. ¿Qué sucede? ¿Cuál es el problema? ¿Cómo se soluciona?

- Problema con el paso de parámetros
    - Vuelva a editar los códigos anteriores y fíjese en el paso de parámetros. ¿Qué espera recibir la subrutina sub2, parámetros pasados por valor o por referencia? ¿Qué implica esto?
    - Pruebe a cambiar el paso de parámetros intentando hacerlo por valor. Compile. ¿Puede compilar? Ejecute. ¿Qué sucede?

LAB 2. Aritmética del computador

4.- Optimización de las operaciones en punto flotante

- Échele un vistazo a ejercicio4.c. Este ejercicio realiza operaciones en punto flotante. En concreto, este ejercicio realiza una división por un valor constante dentro de un lazo. Una de las posibles optimizaciones, dado que la división es una operación costosa, consiste en sacar fuera del lazo la constante 1/E. El código resultante lo puede ver en ejercicio4a.c. Compile (nivel +O0) ambos códigos con E=1.234 como valor. Ejecute y mida ambos tiempos de ejecución. ¿Qué observa?

5.- Problemas de redondeo, rango y precisión

- Compile ahora los códigos anteriores con E=1.234E307. Ejecute los programas y mida su tiempo de ejecución. ¿Se ha parado a mirar los resultados de cada una de las versiones? ¿Qué observa?

Valor E	Programa	Tiempo ejecución	Tamaño ejecutable	resultado
E=1.234	*Ejercicio4*			
	Ejercicio4a			
E=1.234E307	*Ejercicio4*			
	Ejercicio4a			

- Eche un vistazo al código del ejercicio5.c. Compílelo y ejecútelo. El error es mucho más elevado de lo que cabría esperar. ¿Por qué?
- En el código ejercicio5b.c se calcula el mismo valor de dos formas diferentes. Compile y ejecute el código, y comprueba que no se obtiene el mismo valor en los dos casos. ¿Por qué?

6.- Optimización automática en códigos que manejan punto flotante

- Compile el código ejercicio4.c probando los diferentes niveles de optimización. Para todos ellos ejecute y compare los tiempos de ejecución. ¿Qué observa? Fíjese también en el resultado de cada uno.

Optimización	Tiempo compilación	Tamaño ejecutable	Tiempo ejecución	resultado
+O1				
+O2				
+O3				
+O4				
-O				

LAB 3. Uso de librerías

7.- Uso de las librerías BLAS y LAPACK

- Edite los ficheros mxv.f y daxpy.f. El primero de ellos contiene la subrutina mxv que realiza un producto matriz-vector. El segundo de ellos contiene una subrutina (axpy) que realiza la operación Y=Y+aX. Ambas son dos operaciones muy comunes en códigos matemáticos. El fichero ejercicio7.f contiene un bloque main que usa ambas subrutinas. Compile los tres ficheros y ejecute el programa. Mida el tiempo de ejecución.
- La librería BLAS nos permite usar las funciones mxv y axpy ya implementadas:

  call dgemv (trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
  call daxpy (n, a, x, incx, y, incy)

  Copie ejercicio7.f en ejercicio7a.f y edítelo. Sustituya la llamada a mxv por la llamada a la función BLAS correspondiente. Compile de nuevo (esta vez recuerde que debe enlazar con la librería BLAS) y ejecute. Mida el tiempo de ejecución. ¿Qué observa?
- Sustituya ahora la llamada a daxpy del ejercicio7.f por daxpy de BLAS (ejercicio7b.f). Compile y ejecute. Mida el tiempo de ejecución.
- Sustituya ambas llamadas a mxv y axpy por sus correspondientes funciones BLAS. Compile, ejecute y mida tiempo de ejecución. ¿Qué conclusión extrae de todo esto?

Funciones	**Tiempo de ejecución**	**Tamaño del ejecutable**
mxv + axpy		
dgemv(blas) + axpy		
mxv + daxpy(blas)		
dgemv(blas) + daxpy(blas)		

- En el fichero sgesv encontrará un programa que resuelve un sistema de ecuaciones lineales de la forma Ax=b usando LAPACK. Échele un vistazo, compílelo y ejecutelo.
- En el fichero zgeev encontrará un programa en C que encuentra los autovalores de una matriz compleja. Edítelo y fíjese en la llamada a la función de LAPACK correspondiente. ¿Qué observa?

8.- Creación de librerías propias: librerías estáticas

- Usando como base los ficheros mxv.f y axpy.f del ejercicio anterior, compílelos creando una librería estática: libmate.a
- Compile y enlace el programa ejercicio7.f con la librería libmate.a (utilice la opción de compilación –l para indicar la librería). Llame al ejecutable ejercicio8, para no sobrescribir los anteriores
- Ejecute el programa y mida el tiempo de ejecución. Compárelo con los tiempos obtenidos antes.

- Mida el tamaño del ejecutable.

Librería	Tamaño	Tiempo de compilación	Tiempo de ejecución
Libmate.a			

9.- Creación de librerías propias: librerías dinámicas

- Usando como base los ficheros mxv.f y axpy.f, compílelos creando una librería dinámica: libmate.so
- Compile y linka el programa ejercicio7.f con la librería libmate.so. Llame al ejecutable ejercicio9, para no sobrescribir los anteriores
- Ejecute el programa y mida el tiempo de ejecución. Compárelo con el del ejercicio anterior
- Mida el tamaño del ejecutable. ¿Qué observa?

Librería	Tamaño	Tiempo de compilación	Tiempo de ejecución
Libmate.so			

10.- Librerías estáticas vs dinámicas

- Borre o mueva de directorio la librería libmate.a. Vuelva a ejecutar (sin recompilar) el programa ejercicio8. Mida el tiempo de ejecución.
- Ahora borre (o mueva de directorio) la librería libmate.so. Vuelva a ejecutar (sin recompilar) el programa ejercicio9. ¿Qué ha pasado?

11.- Creación de ficheros make

- En el directorio saludo encontrará dos ficheros C: main.c y rut.c. En línea de comandos la compilación sería muy sencilla:

  cc –o hola main.c rut.c.

  Sin embargo, queremos construir un fichero make que haga esto mismo. Siga los siguientes pasos:
  - Objetivo primario: crear el ejecutable (hola)
  - Dependencias: los archivos objeto (main.o rut.o)
  - Regla para crear el objetivo a partir de las dependencias:
    - cc –o hola main.o rut.o
  - Objetivos secundarios: crear cada uno de los objetos (main.o y rut.o)
  - Dependencias: sus respectivos fuentes (main.c y rut.c)
  - Regla para crear el objeto a partir de sus dependencias:
    - Comando de llamada al compilador con la opción –c
- El make anterior puede ser simplificado. Fíjese que si en lugar de tener dos ficheros tenemos 20 o 30, la tarea de escribir el make anterior sería demasiado pesada. Simplifique el fichero make anterior usando una regla de sufijos que transforme todos los .c en .o automáticamente.
- Añada macros al make anterior de forma que simplifique una posible actualización futura. Por ejemplo, use CC = cc de forma que en el futuro si decide cambiar el compilador de C (por ejemplo usar gcc) no tenga que cambiar todas las líneas del make donde se use ese compilador.

LAB 4. Optimización del rendimiento de la jerarquía de memoria

12.- Estudio del efecto de las transformaciones en los códigos.

Aunque el compilador lo puede hacer por nosotros usando las opciones de compilación adecuadas, es bueno tener una idea básica de lo que está haciendo. El objetivo de los siguientes ejercicios es ver como afectan ciertas transformaciones sencillas al rendimiento de nuestro código.

Échele un vistazo al código ejercicio12.f. Verá que es un código que realiza mucho trabajo para hacer algo muy sencillo. Vamos a optimizarlo manualmente paso a paso:

Original:

- Compile el código ejercicio12.f , ejecútelo y mida el tiempo de ejecución.

Reduciendo el número de cálculos:

- El mismo programa reduciendo el número de cálculos se puede ver en el fichero ejercicio12a.f. Compílelo, ejecútelo y mida el tiempo de ejecución. ¿Qué observa?

Evitando llamadas a las subrutinas en el bucle interno:

- En ejercicio12b.f encontrará el mismo programa pero evitando la llamada a la subrutina en el bucle interno. Esta subrutina es muy pequeña, y se llama muchas veces, podemos pensar que compensa hacer *inlining*.
- Compile este código, ejecútelo y, de nuevo, mida el tiempo de ejecución. ¿Compensa?

Evitar llamadas a subrutinas en el bucle externo:

- Repita los pasos anteriores para el bucle externo. Cree usted mismo el fichero ejercicio12c.f que contemple esta optimización.
- Compile, ejecute y mida el tiempo de ejecución.

Fusión de bucles:

- El código ejercicio12d.f contiene el mismo código pero fusionando los bucles. Compile, ejecute y mida el tiempo.

Eliminar resultados intermedios:

- De nuevo en ejercicio12e.f encontrará el código anterior optimizado, esta vez eliminando resultados intermedios. Compile, ejecute y mida el tiempo de ejecución.

Programa	Tiempo ejecución	Tamaño ejecutable	resultado
Ejercicio12			
Ejercicio12a			
Ejercicio12b			
Ejercicio12c			
Ejercicio12d			
Ejercicio12e			

13.- El código del ejecercicio13.f multiplica matrices usando la variante ijk en el acceso a los datos.

- Codificar la multiplicación usando la variante jki. Compilar, ejecutar y medir tiempos para comparar ambas versiones.
- Aplicar la técnica de bloqueo de matrices a la versión jki. Compilar, ejecutar y medir tiempos para comparar ambas versiones

14.- Optimizar el código del ejercicio14.f utilizando los siguientes pasos:

- Intercambiar los lazos.
- Eliminar invariantes.
- Desenrollo de lazos.

Compilar, ejecutar y medir tiempos después de cada optimización.

15.- Optimización automática usando el compilador

- Compile los códigos de los ejercicios anteriores probando los diferentes niveles de optimización del compilador. Para todos ellos ejecute el código resultante y compare los tiempos de ejecución. ¿Qué observa?

LAB 5. Depuración y evaluación del rendimiento

16.- Uso del depurador GDB

- En el directorio GDB1 encontrará los códigos ejercicio16a.f y ejercicio16b.c. Compílelos y ejecútelos. ¿Qué sucede?
- Utilice el depurador gdb para ejecutar cada uno de los códigos paso a paso y detectar donde está el problema. Siga las siguientes indicaciones:
    - Con el código ejercicio16a.f ejecute paso a paso hasta detectar el momento en que se produce un fallo
    - Repita la operación fijándose en la evolución de las variables
    - Corrija el error, vuelva a compilar y ejecute de nuevo.
    - Con el código ejercicio16b.c pruebe a colocar puntos de ruptura (breakpoints) en lugares donde las variables puedan tomar valores conflictivos. Por ejemplo, compruebe si la variable E toma el valor 0 antes de ser usada como divisor. Una división por cero produciría un error.
- En el directorio GDB2 encontrá dos ficheros operaciones.f y main.c. Écheles un vistazo. El fichero main.c contiene el programa principal al que se le pasan dos enteros como argumentos y realiza una serie de operaciones llamando a las subrutinas correspondientes en Fortran. Compile, ejecute y compruebe el resultado. ¿Qué sucede?
- Utilice el depurador gdb para encontrar el problema. Fíjese en el valor de las variables en cada paso, o acote la búsqueda con breakpoints si sospecha donde está el error. Resuelva el problema. Compile de nuevo y ejecute.

17.-Evaluación del rendimiento: medidas de tiempo

- Hasta ahora hemos venido utilizando el comando time para medir los tiempos de ejecución de nuestros programas. Ahora vamos a probar otras medidas de tiempo internas al código. Tome dos cualesquiera de los códigos de los ejercicios que hemos hecho durante el curso, uno en C y otro en Fortran. Introduzca las funciones de medida de tiempos adecuadas. Mida la ejecución completa del programa (desde principio a fin) y la ejecución de una sección del mismo. Compile y ejecute. Mida también el tiempo de usuario con el comando time. ¿Qué observa?

18.-Evaluación del rendimiento: gprof

- El primer ejemplo que vamos a tratar usa un programa muy sencillo “simple.f”. Este programa consta de dos subrutinas. Una de ellas simplemente realiza una operación de suma sobre dos parámetros que se le pasan.
    - Compila este programa con la opción –G para poder obtener después su perfil
    - Ejecuta el programa
    - Ahora tendrá en el directorio un fichero gmon.out. Este fichero contiene el perfil estático de su ejecución, sin embargo no podemos leerlo directamente. Necesitamos usar la utilidad gprof que nos generará un

informe sobre este perfil. Redireccione el informe al fichero simple.profile
    - Observe la información que proporciona el informe. ¿Cuántas veces se ha llamado a la subrutina suma? ¿Cuánto tiempo hemos "perdido" en esa subrutina?
    - ¿Sería una buena idea hacer inlining de esta subrutina? Probémoslo. Cambie el fichero simple.f para hacer la suma directamente en lugar de llamar a la subrutina suma. Compile, ejecute y genere el informe sobre el perfil. Observe el nuevo perfil. ¿Ha habido suerte?

- El segundo ejemplo usa dos versiones simples del mismo programa, una versión llamada untuned y otra llamada tuned. Puede encontrar en su directorio las versiones para C y para Fortran. Use la que le sea más familiar. Vamos a usar gprof para demostrar como testear los esfuerzos que realizamos a la hora de optimizar nuestro código.
    - Examine la versión "untuned" del programa. Observe que consta de cuatro rutinas llamadas: pipe, unroll, strength y block
    - Compile el programa asegurándose de usar la opción –G, que permitirá sacar posteriormente el perfil
    - Ejecute esta versión
    - Redireccione el informe al fichero untuned.profile
    - Examine este fichero untuned.profile y fíjese en los tiempos de ejecución de cada subrutina. Observe también que aparecen librerías del sistema en el informe.
    - Ahora compile la versión "tuned" de su código.
    - Ejecute esta versión generando así el perfil. Ejecute gprof para generar el informe sobre el perfil y examínelo.
    - Compare los tiempos de las rutinas en los dos informes. Con la excepción de la rutina pipe, la versión "tuned" ofrece mejor rendimiento. Los comentarios en esa versión explican brevemente que se ha hecho para mejorar el rendimiento.
    - Como comentábamos en el tema 1, el compilador puede realizar muchas optimizaciones por nosotros. Compile la versión "untuned" con distintos niveles de optimización y genere sus perfiles. Observe los informes y compare los resultados.