Computación Paralela 2019/2020
Memoria de la Práctica 2: MPI

Grupo: g110
Alumnos: Manuel de Castro Caballero, Pablo Martínez López

1. Puntos fundamentales y soluciones aplicadas para conseguir el objetivo.

Inicialmente tratamos de paralelizar los vectores de *culture* y *culture_cells*, es decir, partir las estructuras y distribuirlas entre varios procesos, y con el transcurrir de la práctica nos dimos cuenta de que también sería conveniente distribuir la lista de células. Pudimos partir correctamente las dos primeras estructuras (línea 507) simplemente dividiendo su tamaño entre el número de procesos (Fig. 1 y 2), y calculando si las células generadas correspondían o no al sub-vector correspondiente a un proceso dado, según su *rank* (línea 374).

También realizamos ciertas modificaciones en el código secuencial, bien por ideas propias así como inspiradas en códigos de otros compañeros de la práctica de OpenMP. La más notable consiste en separar los bucles de la simulación en 2: uno para las 10 primeras iteraciones (línea 818), que son especiales ya que hay condiciones que nunca se van a cumplir y no hace falta comprobarlas; y otro para el resto de iteraciones; de haberlas (línea 1009).

Como la lista de células está distribuida entre varios procesos, cuando se mueve una célula puede que debamos enviarla a otro proceso y eliminarla del primero. Para ello añadimos un bloque nuevo, el bloque *4.X* (línea 1115), que mediante comunicaciones del tipo *all to all* enviamos a todos los procesos el número de células que va a recibir primero, y luego las células en sí.

Con respecto a comunicaciones, también añadimos reducciones en las secciones finales del código de la simulación (línea 1284), para que todos los procesos sepan si se debe o no iterar de nuevo, y para recolectar la información de salida del programa. Como las comunicaciones entre procesos son lo que más tiempo consume en el programa, intentamos mantener el número de comunicaciones al mínimo, llegando a juntar varias comunicaciones (reducciones) de datos distintos en una única comunicación de un array con dichos datos (línea 1240). Con esto llegamos a los 54 segundos de tiempo de ejecución en el tablón.

En un punto nos planteamos pasar a realizar pruebas con el reparto de trabajo entre distintos procesadores/máquinas. Tras realizar algunas pruebas, finalmente comprobamos que ejecutar el programa finalizando directamente los procesos de *heracles* nada más identificarlos (línea 459) disminuía considerablemente el tiempo de ejecución del mismo. Entendemos que esto se debe a que la red del cluster del tablón es muy lenta; así como a que, de entre las dos máquinas, *hydra* es la más rápida. Con la adición de esta relativamente sencilla característica, logramos superar directamente la segunda referencia, con un tiempo de 38 segundos.

Lo último que se nos ocurrió fue aplicar paralelismo de tareas al bloque *4.1* (línea 549), ya que es una de las secciones que más tiempo consumen en el programa, y es inherentemente secuencial. Dedicamos un proceso exclusivamente a realizar este bloque y comunicar los resultados a los demás procesos cuando la requieren, optimizando las comunicaciones, de tal forma que este proceso esté bloqueado el menor tiempo posible (Fig. 3). Esto redujo nuestro tiempo en la leaderboard a 27 segundos, otorgándonos el primer puesto.

Después, incluimos ciertas mejoras que hacen el programa más robusto (que funcione con 1 proceso, o con más procesos que posiciones de matrices repartir), pero nada que reduzca más el tiempo. El código actual no debería fallar con ningún caso de prueba, en principio (aunque nunca se sabe, pero esperemos que no).

2. Detalles sobre el proceso seguido

Desarrollamos la práctica a través de un servidor propio de Discord, al que tenían acceso cuatro grupos (*g110*, *g211*, *g304* y *g106*). Teníamos un canal de audio para cada grupo, de forma que trabajasen compartiendo su pantalla. También creamos canales de texto en los que copiamos mensajes importantes del grupo de Discord de la asignatura, compartíamos casos de prueba o fragmentos sencillos de código (macros, ejemplos de funciones de *MPI*...), y preguntábamos y resolvíamos dudas entre nosotros. De vez en cuando compartimos ideas, o recomendaciones para solucionar algún problema.

En el caso de nuestro grupo, utilizamos un repositorio *git* para desarrollar la práctica, y por lo general quedábamos a trabajar los dos al mismo tiempo, compartiendo uno la pantalla, y el otro ayudando con documentación. En muy raras ocasiones se trabajó por separado. A la hora de debuggear, cada uno trabajaba sobre el código de forma individual, comentando los resultados. Normalmente el proceso de *debugging* se realizaba "a lo bruto", imprimiendo valores de variables en el código, aunque también llegamos a utilizar *Valgrind*, sobre todo los últimos días.

En cuanto a casos de pruebas, tenemos una colección general bastante amplia de 60 casos, de entre los que podemos destacar:

./evolution 29 40 600 1000.0 0.005 5.0 22177 37626 24340 8 5 5 12 12 0.05 12

./evolution 2900 4000 1000 3000.0 0.015 12.0 0 0 0 32 100 100 100 100 0.31 30

./evolution 2000 2000 300 1000.000000 0.005000 5.000000 28783 648 47734 8 5 5 12 12 0.050000 12.000000

./evolution 10000 10000 10000 10000 0.01 5 22177 37626 24340 10

./evolution 3 2 1 100.0 0.011 15.0 22177 37626 24340 8

3. Material previo utilizado

Partimos de nuestro código de OpenMP, así como de los mejores códigos en la leaderboard para esa práctica, especialmente el del grupo *g210*. Reutilizamos las pruebas desarrolladas para la práctica anterior, y añadimos otras desarrolladas por compañeros (como las del grupo *g304*).

En varias ocasiones utilizamos las soluciones de los ejercicios de laboratorio para implementar alguna funcionalidad en la práctica. También pedimos algún consejo a veteranos, pero tampoco se acordaban de demasiado. En alguna ocasión, consultamos los programas desarrollados por algún veterano en su respectivo curso, y que habían tenido a bien dejar públicos en repositorios *git*; aunque el programa es lo suficientemente distinto como para no sacar demasiado en claro.

Evidentemente, también revisamos las transparencias de la asignatura antes de desarrollar soluciones.

4. Aportaciones personales

Como hemos indicado previamente, el desarrollo de la práctica se realizó en su inmensa mayoría en conjunto, mediante llamadas de Discord. Por tanto, no hay ninguna característica que haya sido implementada exclusivamente por uno de los miembros; aunque sí que podemos destacar quién tuvo ciertas ideas:

Pablo: se encargó sobre todo de las partes de comunicación de MPI. Estuvo probando diferentes funciones para realizar las comunicaciones entre procesos, así como investigando qué alternativas eran más eficientes.

Manuel: realizó implementaciones en el código para facilitar el desarrollo de la práctica, como las macros o los contadores de tiempo de cada bloque. También realizó las pruebas de reparto de carga entre máquinas (que determinaron que era mejor ejecutar los procesos solo en *Hydra)*, así como la paralelización en un solo proceso del bloque *4.1* (generación de comida), y las mejoras de robustez en el programa.

5. Bibliografía

[1] Software in the Public Interest (SPI), "MPI_Alltoall(3) man page (version 4.0.3) **", Last modification on 4-Mar-2020,** https://www.open-mpi.org/doc/current/man3/MPI_Alltoall.3.php (Utilización de *MPI_Alltoall*)

[2] Software in the Public Interest (SPI), "MPI_Alltoallv (3) man page (version 4.0.3) **", Last modification on 4-Mar-2020,** https://www.open-mpi.org/doc/current/man3/MPI_Alltoallv.3.php (Utilización de *MPI_Alltoallv*)

[3] Woboq, "source code of glibc/stdlib/erand48.c**", 2019-Mar-30,** https://code.woboq.org/userspace/glibc/stdlib/erand48.c.html

6. Figuras

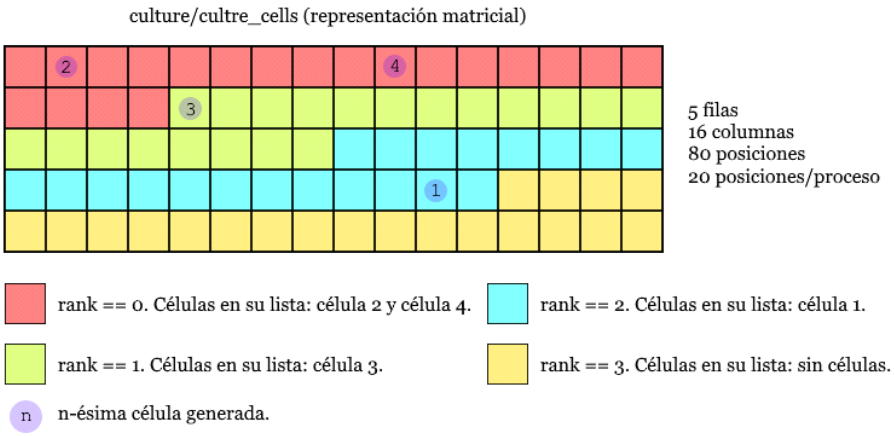

Figura 1. Distribución de las estructuras entre distintos procesos. Ejemplo con 5 filas, 16 columnas y 4 procesos.

Figura 2. Distribución de las estructuras cuando el reparto no es equitativo entre procesos. Ejemplo con 3 filas, 4 columnas y 5 procesos.

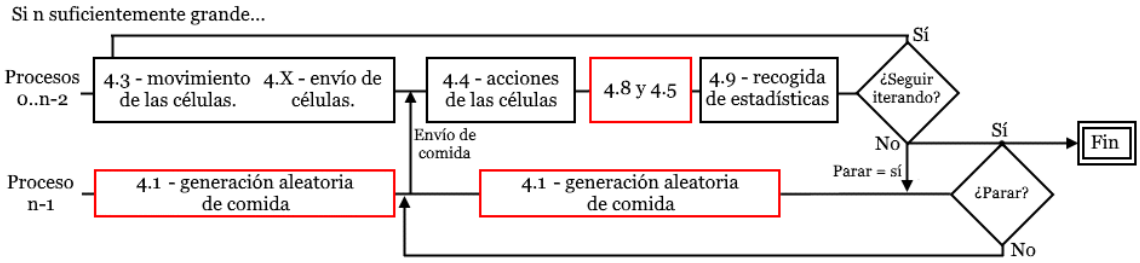

Figura 3. Diagrama simplificado de ejecución del programa en varios procesos distintos. Los bloques en rojo son los que más tardan en ejecutarse generalmente.