Programación con memoria distribuida

Aclaraciones

El presente trabajo ha sido desarrollado en grupo por Franco Emanuel Liptak y Gastón Gustavo Ríos. Consiste en un ejercicio práctico pedido por la cátedra, con dos soluciones diferentes (secuencial y con OpenMPI). El objetivo del trabajo fue resolver el ejercicio con distintas tecnologías, a fin de poder visualizar la mejora del rendimiento de OpenMPI, respecto al algoritmo secencial.

Se ofrece también la posibilidad de descargar los archivos de extensión '.c' (es decir, los archivos que serán utilizados por el compilador, para compilar en base a la gramática y sintaxis de C) desde el siguiente repositorio de GitHub: https://github.com/okason97/sistemasParalelos, el cual por supuesto, tiene como colaboradores a ambos integrantes del grupo.

Nuestros usuarios de GitHub son: - okason97 (Gastón) - FrancoLiptak (Franco)

Las pruebas se hicieron en las computadoras del aula donde cursamos la materia.

Ejercicio - Juego de las N-Reinas

"El juego de las N-Reinas consiste en ubicar sobre un tablero de ajedrez N reinas sin que estas se amenacen entre ellas. Una reina amenaza a aquellas reinas que se encuentren en su misma fila, columna o diagonal. La solución al problema de las N-Reinas consiste en encontrar todas las posibles soluciones para un tablero de tamaño NxN."

Resolución secuencial

Para ejecutar el script el usuario debe ingresar como parámetro un valor N, siendo N el tamaño del tablero de NxN donde se ubicarán las reinas.

A continuación describiremos el funcionamiento del script:

Primero se declaran e inicializan las variables necesarias, y se alocan los vectores a utilizar.

Los vectores que utilizaremos serán `queens`, `col_available`, `asc_diagonal` y `des_diagonal`:

- En el vector `queens` se almacenarán las posiciones de las reinas, dado que según las reglas impuestas solo podrá haber una reina por fila. Se utiliza el índice del vector de `queens` como indicador de fila y el valor como indicador de columna. De este modo, `queens[i] = j` indicaría que la reina está posicionada en la posición (i,j).
- Los vectores `col_available`, `asc_diagonal` y `des_diagonal` aseguran de forma eficiente que las reinas no se interfieran entre si. Cada vez que una reina se posiciona reservará las 2 diagonales y la columna para que ninguna reina pueda posicionarse en éstas. Esto permite verificar que una columna para una reina dada sea válida en 1 acceso por vector.

Se puede pensar el problema como N árboles, con N siendo la cantidad de filas o columnas del

tablero. Cada posición de fila de una reina es un nivel del árbol, comenzando con la posición de fila 0 como la raíz, la cual tendrá en cada árbol un valor `j` diferente, siendo `j` la columna en la que se posiciona la reina. De esta raíz se desprenden las posibles variaciones de `j` válidas de la reina posicionada en la siguiente fila, y esto se repite hasta llegar a la fila N, en cuyo caso se ha encontrado una solución, o hasta llegar a una fila en la cual no existen posiciones de columna válidas.

Nuestra solución realiza el recorrido de este árbol construyendo y deconstruyendo el árbol en cada iteración y contando la cantidad de veces en las que se llegó a la fila N (cantidad de soluciones posibles).

Para el posicionamiento de las reinas, posicionamos a cada reina `i` en una posición de columna `j` válida. La validez de esta posición `j` está dada por la disponibilidad de una columna (`col_available`), una diagonal ascendente (`asc_diagonal`) y una diagonal descendente (`des_diagonal`) que no esté siendo utilizada por ninguna otra reina. Si la reina ya fue posicionada en un paso anterior, esta buscada un valor de `j` superior. De esta forma se evita que la reina pase por valores que ya había utilizado.

Es posible que se llegue a una reina a la cual no se le puede asignar ningón valor de `j`. Esto se debe a que no existe una solución al problema en esa rama o a que ya se han recorrido todos los valores válidos para el nivel del árbol. En ambos casos se debe subir un nivel para buscar otra rama válida. En caso de haberle podido asignar un valor de `j` a la reina en la última posición entonces se aumenta el número de soluciones posibles, ya que se le ha asignado una posición a cada reina.

Finalmente imprimimos el resultado, el tiempo de ejecución y liberamos los vectores alocados.

Resolución con MPI

Para ejecutar el script el usuario debe ingresar como parámetro un valor N, siendo N el tamaño del tablero de NxN donde se ubicarán las reinas.

A continuación describiremos el funcionamiento del script:

Primero se declaran e inicializan las variables necesarias, y se alocan los vectores a utilizar. Los vectores que utilizaremos serán `queens`, `col_available`, `asc_diagonal` y `des_diagonal`, los cuales cumplirán la misma función que en la solución secuencial.

La resolución con MPI está basada en nuestra solución secuencial. La diferencia es que esta vez tenemos que distribuir el trabajo en distintos procesos que van a colaborar entre sí para lograr el resultado esperado en el menor tiempo posible. Para eso, usamos el modelo 'Master-Worker'.

El proceso 0 será nuestro 'Master'/'Coordinador', y se encargará de repartir el trabajo (bajo demanda) a los procesos restantes ('Workers'). Cada proceso 'Worker' va a solitar trabajo al 'Master', el cual puede responder con un valor de columna para la primer reina de una solución. Así, cada proceso trabaja con uno de los árboles que representaría todas las soluciones posibles para un determinado valor de la primer reina. Vale aclarar que el 'Master', además de distribuir el

trabajo entre los 'Workers', también buscará soluciones. Podemos resumir el trabajo del 'Coordinador' de la siguiente manera:

- Luego de la distribución de trabajo inicial, por cada valor de columna aún no asignado el 'coordinador' recibirá de forma no bloqueante a los procesos que hayan terminado su trabajo y requieran de otro valor de columna para la primer reina. Mientras espera que algún worker le solicite trabajo, el 'coordinador' realizará procesamiento utilizando un valor de columna para la primer reina no asignado a un 'worker', lo cual disminuye el desbalance de carga entre los procesos.
- Al finalizar el procesamiento de todos los valores de columna posibles para la primer reina, se le envia a cada proceso el valor -1, el cual utilizarán para determinar que se han terminado de procesar todos los valores de columna posibles para la primer reina.

Los 'workers' realizarán un procesamiento similar al secuencial pero sólo para un valor de columna para la primer reina, y al finalizar el procesamiento requerirán al 'master' (coordinador) otro valor de columna para la primer reina.

Una vez terminado el procesamiento y con los workers avisados de esto, se procede a realizar la reducción de la cantidad de soluciones encontradas (local a cada proceso) a una sola variable.

Finalmente imprimimos el resultado, el tiempo de ejecución local a cada proceso y también el total, y liberamos los vectores alocados.

Mediciones

Tabla con tiempos de ejecucion en secuencial

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones
8	0.000390	92
9	0.001637	352
10	0.007326	724
11	0.029500	2680
12	0.124517	14200
13	0.551008	73712
14	3.312708	365596
15	21.080764	2279184

Tablas con tiempo de ejecución en MPI con 1 computadora y 4 procesos

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones	Tiempo proceso 0	Tiempo proceso 1	Tiempo proceso 2	Tiempo proceso 3	Desbalance de carga	Speedup	Eficiencia
8	0.000224	92	0.000091	0.000078	0.000058	0.00005	0.592057	1,741071	0,435267
9	0.001546	352	0.000993	0.000209	0.000265	0.000292	1.782831	1.058861	0.264715
10	0.002699	724	0.002109	0.001314	0.001206	0.001344	0.604721	2.714338	0.678584
11	0.009596	2680	0.007407	0.006775	0.006064	0.005396	0.313704	3.074197	0.768549
12	0.045124	14200	0.039678	0.030273	0.031206	0.026181	0.423973	2.759440	0.689860

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones	Tiempo proceso 0	Tiempo proceso 1	Tiempo proceso 2	Tiempo proceso 3	Desbalance de carga	Speedup	Eficiencia
13	0.210631	73712	0.183983	0.151058	0.140715	0.126675	0.380511	2.615987	0.653996
14	1.158291	365596	0.991986	0.794524	0.774398	0.810097	0.258187	2.859996	0.714999
15	7.519437	2279184	6.586658	5.372211	5.190085	4.880626	0.309771	2.803502	0.700875

Tabla con tiempo de ejecución en MPI con 2 computadoras con 2 procesos cada una (4 procesos total)

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones	Tiempo proceso 0	Tiempo proceso 1	Tiempo proceso 2	Tiempo proceso 3	Desbalance de carga	Speedup	Eficiencia
8	0.02875	92	0.00007	0.000123	0.000057	0.000027	0.000123	0.013565	0.003391
9	0.027454	352	0.000254	0.000459	0.000258	0.000137	1.162454	0.059627	0.014906
10	0.028889	724	0.001013	0.001621	0.001222	0.000808	0.697255	0.253591	0.063397
11	0.034101	2680	0.004535	0.00696	0.005784	0.00447	0.457952	0.865077	0.216269
12	0.060773	14200	0.022872	0.041675	0.025109	0.024223	0.660455	2.048886	0.512221
13	0.210981	73712	0.150704	0.137845	0.138713	0.139424	0.090766	2.611647	0.652911
14	1.130398	365596	0.958633	0.828178	0.741098	0.730746	0.279731	2.930567	0.732641
15	7.106384	2279184	5.804697	5.39941	5.252491	4.299635	0.290045	2.966454	0.741613

Tabla con tiempo de ejecución en MPI con 2 computadoras con 4 procesos cada una (8 procesos total)

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones	Tiempo proceso 0	Tiempo proceso 1	Tiempo proceso 2	Tiempo proceso 3	Tiempo proceso 4	Tiempo proceso 5	Tiempo proceso 6	Tiempo proceso 7	Desbalance de carga	Speedup	Eficiencia
8	0.045022	92	0.0	0.000078	0.000049	0.000049	0.000029	0.000029	0.000033	0.000035	2.066225	0.008662	0.001082
9	0.047396	352	0.0	0.000279	0.000288	0.000163	0.00018	0.000103	0.000152	0.000139	1.766871	0.034538	0.004317
10	0.047366	724	0.0	0.001057	0.001112	0.001043	0.000445	0.000453	0.000472	0.000391	1.788859	0.154667	0.019333
11	0.046045	2680	0.0	0.004492	0.004826	0.004907	0.0032	0.001851	0.001847	0.001735	1.717385	0.640677	0.080084
12	0.056356	14200	0.0	0.021532	0.022024	0.023107	0.016366	0.01632	0.00851	0.00855	1.587987	2.209471	0.276183
13	0.141969	73712	0.050516	0.096818	0.099207	0.103858	0.09154	0.082561	0.049126	0.045634	0.752175	3.881185	0.485148
14	0.567177	365596	0.263926	0.479148	0.500288	0.501781	0.479738	0.479685	0.41963	0.293252	0.488151	5.840695	0.730086
15	3.236788	2279184	2.515966	2.741121	2.920514	2.946898	2.791373	2.914176	2.836931	1.867824	0.400867	6.512865	0.814108

Conclusión

Gracias al paralelismo utilizando MPI hemos logrado un mejor tiempo que nuestra resolución secuencial en problemas con tableros de tamaño 8 para 4 procesos y 12 para 8 procesos. Este tamaño mínimo se debe al overhead causado por el pasaje de mensajes y la espera sincrónica de algunos de estos.

Pudimos aumentar la eficiencia del programa dándole al proceso maestro trabajo una vez que ya repartió todos los trabajos a los workers, reduciendo el desbalance de carga. También usamos pasaje de mensajes no bloqueantes con el maestro para que no se quede ocioso mientras espera que un worker le pida trabajo. Esto también aumento la eficiencia.

Trabajar con más computadoras nos dió mejor tiempo cuando el problema era de grano grueso (mucho procesamiento y poco pasaje de mensajes) debido a que las capacidades de éstas eran similares y al usar varias se pudo aprovechar la memoria caché de ambos procesadores (favoreciendo mucho procesamiento), y a que el pasaje de mensajes entre computadoras es más

lento que dentro de una misma (desfavoreciendo mucho pasaje de mensajes).

Al utilizar muchos procesos (8) y poco tamaño de tablero se daban situaciones en las cuales la resolucion, debido al tiempo del pasaje de mensaje, daba un speedup menor a 1, pero al aumentar el tamaño del tablero esto se revierte aumentando en gran medida el speedup.