ENTREGABLE PARAL·LELISME

QUART ENTREGABLE

Username: par4105
4 de Desembre de 2018
Tardor del 2018

Àlex Valls
Roger Guasch

Índex

Introducció

L'any 1848, el compositor d'escacs Max Bezzel, va publicar el puzzle de les vuit reines, reptant als jugadors d'escacs com es podrien col·locar exactament vuit reines en un tauler d'escacs sense que cap d'ella estigui en posició d'amenaçar una altre. En altres paraules: el puzzle consisteix en col·locar exactament vuit reines sense que dos d'aquestes estiguin situades en la mateixa línea diagonal, vertical o horitzontal. Dos anys més tard, Franz Nauck va publicar la primera solució i va extendre el problema a n reines, situades en un taulell de mida n x n.

En informàtica, el problema de n reines, es resol utilitzant un algorisme de backtracking o força bruta. El sistema prova totes les possibles combinacions de n reines en un tauler de mida n x n i mostra totes les possibles solucions. En cas que només es desitgi obtenir una solució possible, l'algorisme es pot implementar amb un mecanisme de cut-off o poda, que aturi l'execució del backtracking.

Per aquestes dues sessions de laboratori hem treballat amb una implementació de backtracking del problema amb llenguatge C. L'objectiu és realitzar un petit estudi sobre el codi, utilitzant les eines disponibles, i aconseguir una bona paral·lelització de la funció de backtracking per millorar el temps d'execució.

Understanding the potential parallelism in Nqueens

Per realitzar la primera part d'aquest estudi vam començar per analitzar el codi de l'arxiu nqueens.c per tal d'incloure les crides de de la API del Tareador. El programa estava escrit amb compilació condicional, fet que implica que algunes parts escrites del nostre codi seran o no compilades en funció de quina opció seleccionem quan escrivim la comanda make a la terminal. Per exemple, en la Figura 4.1 es pot veure la comanda utilitzada per obtenir la versió seqüencial i, en la Figura 4.2, la comanda utilitzada per obtenir la versió paral·lela.

```
par4105@boada-1:~$ make nqueens-seq
```

Figura 4.1 Comanda executada per compilar el arxiu nqueens.c en versió seqüencial

```
par4105@boada-1:~$ make nqueens-omp
```

Figura 4.2 Comanda executada per compilar el arxiu nqueens.c en versió paral·lela

Per tal d'obtenir la versió necessària per veure el graf en el programa Tareador, la comanda executada va ser la de la Figura 4.3. Cal tenir en compte que, un cop compilem l'arxiu amb l'opció -tar, tot el codi incluit dins la regió *#ifdef _TAREADOR_* serà compilat per aquesta versió en concret.

```
par4105@boada-1:~$ make nqueens-tar
```

Figura 4.3 Comanda executada per compilar el arxiu nqueens.c en versió vàlida per utilitzar el Tareador

Així doncs, al realitzar aquesta compilació, estarem realitzant les crides TAREADOR_ON() i TAREADOR_OF(), per declarar una regió on ha d'actuar el Tareador, així com també estarem compilant les crides *tareador_start_task()* i *tareador_end_task()*. Després d'obtenir la versió desitjada, vam executar el run-tareador.sh per obtenir el gràf que veurem a continuació, en la Figura 4.4.

Aquest graf representa les tasques que realitza un programa simulant una solució en un taulell de tamany 4 x 4. Com es pot apreciar, el programa és capaç de trobar una possible solució i generar el graf de tasques corresponent. Si ens fixem, veurem com existeix una clara dependència degut a l'accés a les variables *sol* i *sol_count*, fet que origina que el graf de la Figura 4.4 tingui aquesta forma. En concret el graf troba dues possibles solucions, com es pot observar si es busca la tasca *solution*, representada per una rodona blava en el graf. La primera solució es troba, dins del primer rectangle verd, en un rectangle de color més verd marí. La segona i última solució es pot localitzar dins del quart rectangle, el que és de color vermell i està situat a la part inferior dreta del graf.

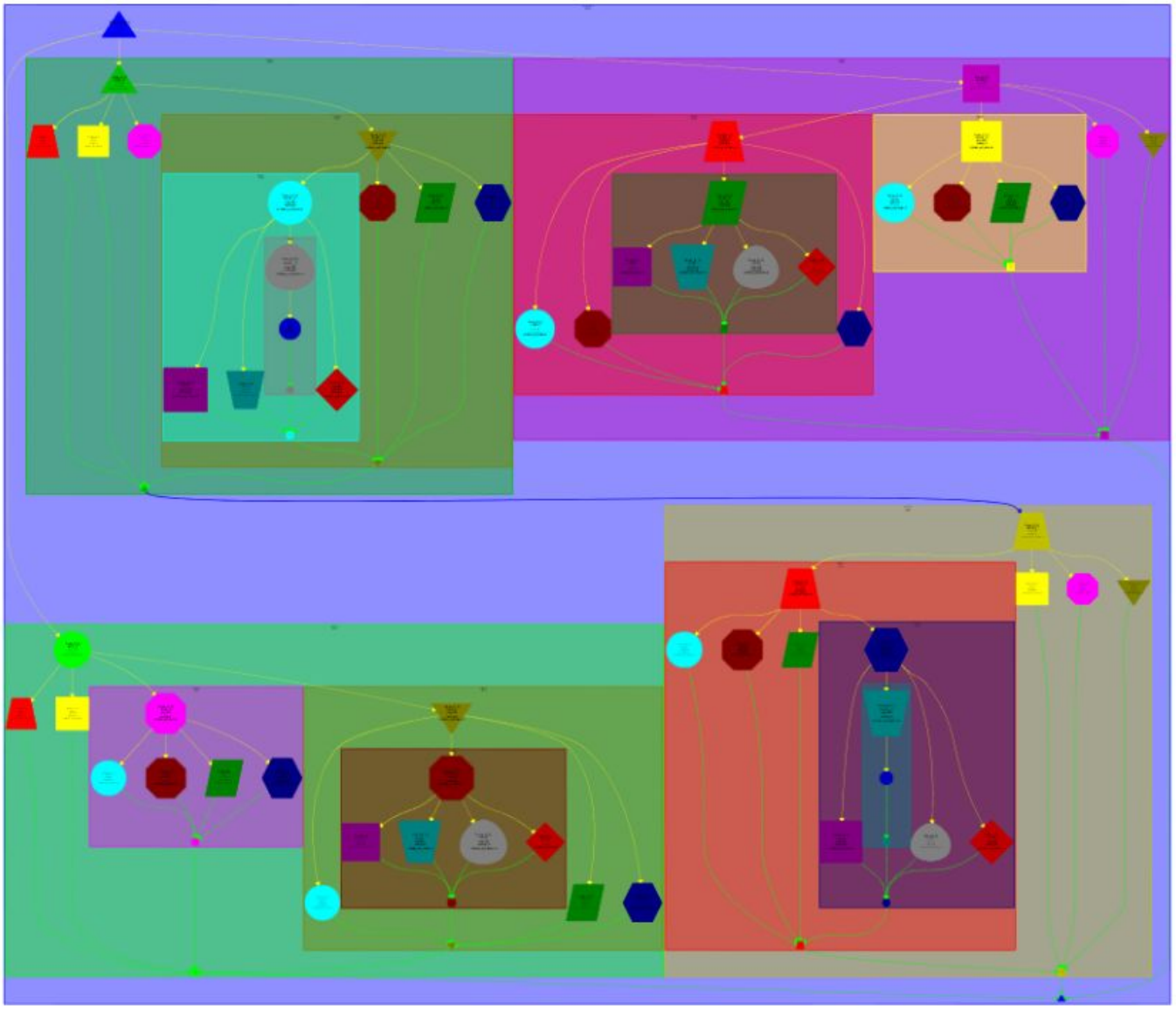

Figura 4.4 Graf obtingut amb el Tareador, utilitzant l'opció nqueens-tar a l'hora de compilar

Un cop ja som conscients d'aquestes dependències podem fer que el tareador ens les deixi de mostrar. Per aconseguir-ho cal escriure en el codi les dues crides a la API del tareador de la Figura 4.5, passant per paràmetre les variables que originen les mencionades dependències.

```
tareador_disable_object(sol);
tareador_disable_object(sol_count);
```

Figura 4.5 Crides a la API per aconseguir que el tareador no ens mostri les dependències

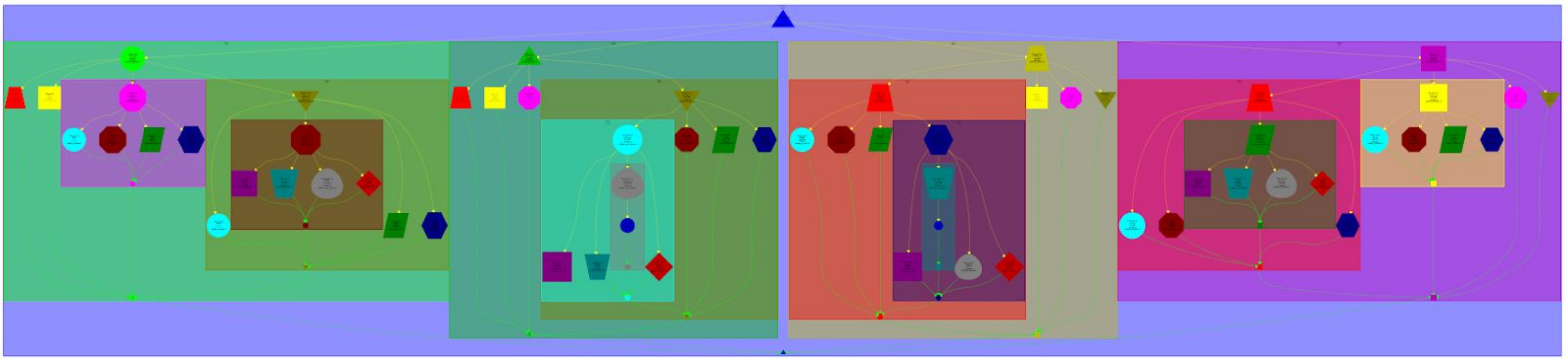

Figura 4.6 Graf obtingut amb el Tareador, utilitzant l'opció nqueens-tar a l'hora de compilar, un cop eliminades les dependències causades per les variables sol i sol_count

Repetim el mateix procés que hem fet, utilitzant la mateixa opció -tar a l'hora de compilar, per obtenir el graf de la Figura 4.6 després d'executar el script corresponent. Com es pot veure, ara ja no ens indica les dependències originades per les variables sol i sol_count, tot trobant les mateixes solucions que abans.

Shared-memory parallelization

En aquesta segona part de la sessió hem treballat amb la paral·lelització del codi de *nqueens.c.* Per aconseguir aquest objectiu ha sigut necessari incloure les directives de la Figura 4.7 en la funció *main* del nostre codi. Aquestes dues directives declaren una regió paral·lela i evita que més d'un thread crei tasques, per evitar duplicitats.

```
#pragma omp parallel
#pragma omp single
```

Figura 4.7 Directives OpenMP afegides a la funció main del nostre codi

També ha sigut necessari modificar la funció *void nqueens*. El codi d'aquesta es pot trobar en la Figura 4.8. Per tal d'aconseguir una bona paral·lelització ha sigut necessari modificar el codi de la funció. Hem protegit amb un *critical* el tros de codi encarregat de reservar memòria, un cop la funció troba una possible solució. Era necessari evitar que dos threads poguessin reservar el mateix espai de memoria, en el mateix temps, per emmagatzemar dues solucions diferents.

```
void nqueens(int n, int j, char *a,int depth) {
    int i;
    if (n == j) {
        if( sol == NULL ) {
            #pragma omp critical
            {
                    sol = malloc(n * sizeof(char));
                    memcpy(sol, a, n * sizeof(char));
                    }
            }
            #pragma omp atomic
            sol_count += 1;
    } else {
        for ( i=0 ; i < n ; i++ ) {
                a[j] = (char) i;
            if (ok(j + 1, a)) {
                                char *ptr = malloc(n * sizeof(char));
                                memcpy(ptr, a, n*sizeof(char));
                                #pragma omp task  final(depth >= cutoff) mergeable
                                {
                                nqueens(n, j + 1, ptr, depth+1);
                                        free(ptr);
                    }
                }
        }
    }
}
```

Figura 4.8 Funció nqueens paral·lela

Hem trobat necessari protegir amb un *atomic* l'increment de la variable *sol_count* ja que, si no estigués protegida, dos threads podríen augmentar la variable al mateix temps i obtindriem valors erronis.
Seguidament hem implementat la optimització comentada a classe per tal de reduir el temps que es triga en accedir al punter *a* copiant les dades a un punter auxiliar, anomenat *ptr*. Cal també declarar una tasca paral·lela amb la directiva *task* del OpenMP i alliberar la memòria ocupada pel punter auxiliar. Per implementar una versió tree amb cut-off cal que, al escriure la directiva *task*, també especifiquem un *final* per evitar la crear més tasques de les que poden ser processades durant l'execució del programa.

El cut-off afecta directament al temps total d'execució tal i com es pot comprovar a la Figura 4.9, els resultats de la qual s'han obtingut gràcies a l'execució del script submit-omp.sh.

Nivell de Cut-Off	Temps d'execució (s)
3	0.287923
4	0.276375
5	0.340276
6	0.350401

Figura 4.9 Taula on es mostra, per diferents valors de cut-off, el temps d'execució obtingut

Així doncs, pel nostre codi en concret, un cut-off de 4 o 5 és un bon valor ja que, quan el codi deixa de generar tasques paral·leles a partir del quart o cinquè nivell de l'arbre, obtenim els millors temps d'execució.

Seguidament, amb la comanda de la Figura 4.10, varem obtenir les gràfiques de *speed up* i *execution time* del nostre programa. Els resultats obtinguts es poden apreciar a continuació.

```
#pragma omp parallel
#pragma omp single
```

Figura 4.10 Comanda per obtenir les gràfiques de execution time i speed up del nostre programa

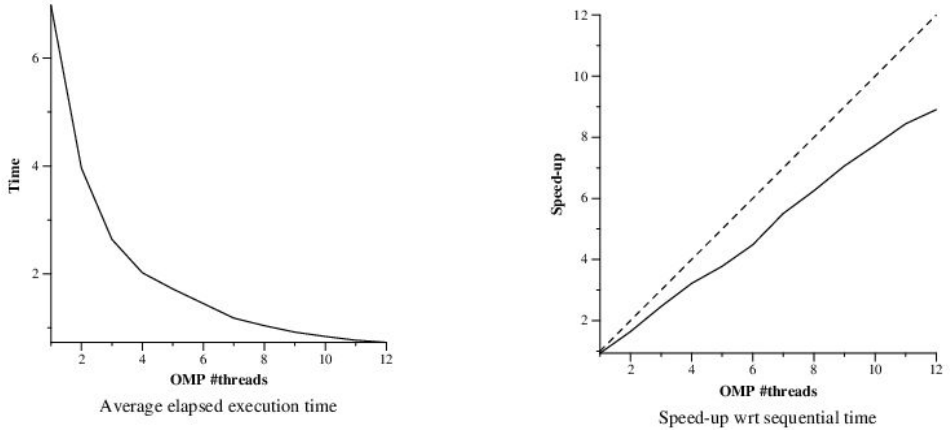

Figura 4.11 Gràfiques de temps d'execució i speed up del nostre programa

Aquestes gràfiques les hem obtingut amb un *cut-off* de 4. Es pot veure clarament com, a mesura que utilitzem més threads, el temps d'execució del programa disminueix de forma exponencial. Com era d'esperar, obtenim grans resultats per al speed-up.
Vam obtenir també altres gràfiques amb valors diferents de cut-off, modificant el script *submit-strong-omp.sh*. Les gràfiques de execution time obtingudes eren força similars a les que hem obtingut per a cut-off amb valor 4 però, per contra, el *speed-up* era inferior, així que vam comprovar que, efectivament, el 4 es un bon valor per al *cut-off*.

Per acabar el nostre estudi vam obtenir les traces amb el *paraver*. Tot i tenir certs problemes amb el boada, degut a que teniem moltes traces d'altres laboratoris ocupant l'espai del disc, vam aconseguir generar els gràfics de la Figura 4.12 i la Figura 4.13 eliminant arxius innecessaris.

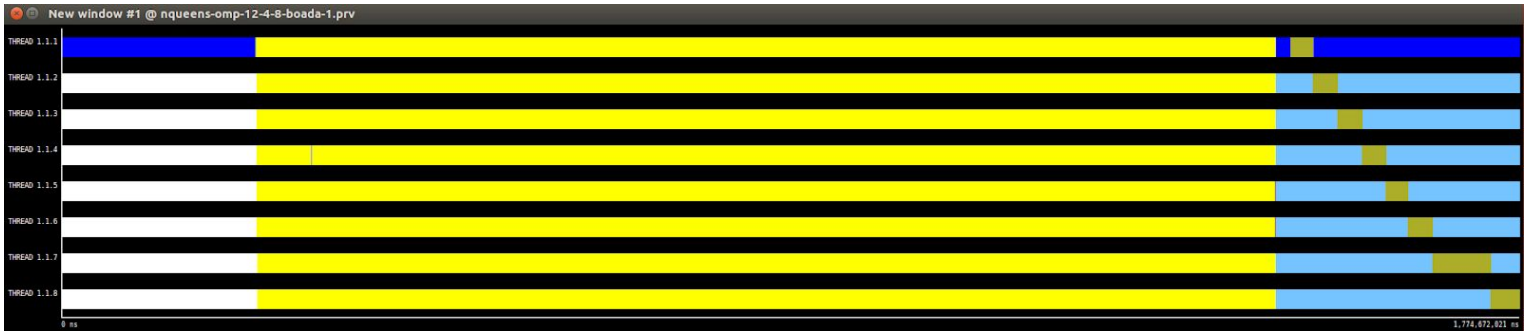

Figura 4.12 Traça completa de l'execució de nqueens amb un taulell de mida 12 x 12

La conclusió que arribem és que el primer thread necessita més de tres quartes parts del temps d'execució del programa per poder generar les tasques que, posteriorment, els altres threads s'encarregaran d'executar.
En aquesta figura no es pot apreciar on comença la feina en paral·lel, així que vam decidir fer zoom-in en la zona on finalitza el color groc -scheduling- i la que comença, de nou, el color blau fosc del primer thread.

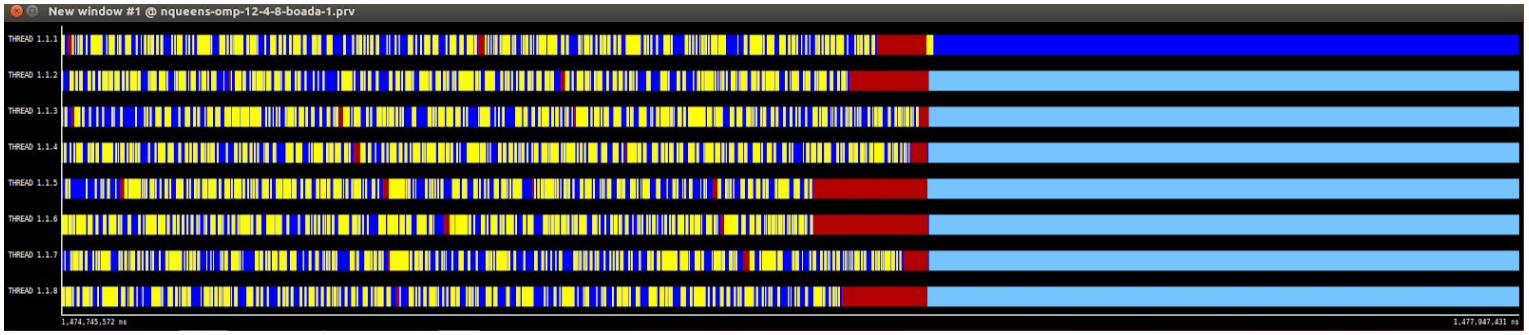

Figura 4.13 Zoom-in de l'execució de nqueens amb un taulell de mida 12 x 12

D'aquesta traça podem extreure que, el temps que el programa s'està executant en paral·lel, és molt inferior respecte al temps que s'executa de forma seqüencial. També cal mencionar que les tasques en paral·lel generades son poc intensives, ja que cada una d'elles ha de realitzar moltes poques instruccions de càlcul complex.

Opcional

En aquest apartat opcional es demana canviar la versió del codi paral·lel que hem fet en els apartats anteriors per un codi amb paral·lelització de bucle, és a dir, utilitzant #pragma omp for. El resultat d'aquesta nova versió es pot veure en la Figura 4.14.
Per tant, hem esborrat el #pragma omp task i el taskwait, i hem afegit la instrucció *#pragma omp parallel for shared(a) schedule (static, n/numero_threads)* just a sobre el for que recorre les n posicions. D'aquesta manera paral·lelitzem cada iteració del bucle i per tant les exploracions en les n posicions en cada columna. Posem shared(a) perquè tots els threads comparteixin a i utilitzem scheduling static dividint les n iteracions per el nombre de threads. Fent aquesta paral·lelització obtenim uns resultats molts similars als de la versió anterior.

```
void nqueens(int n, int j, char *a,int depth) {
    int i;
    if (n == j) {
            /* put good solution in heap. */
        if( sol == NULL ) {
            #pragma omp critical
            {
                    sol = malloc(n * sizeof(char));
                    memcpy(sol, a, n * sizeof(char));
                    }
            }
            #pragma omp atomic
            sol_count += 1;
    } else {
        /* try each possible position for queen <j> */
            int num_t = omp_get_num_threads();
            #pragma omp parallel for shared(a) schedule(static,n/num_t)
        for ( i=0 ; i < n ; i++ ) {
              a[j] = (char) i;
            if (ok(j + 1, a)) {
              char *ptr = malloc(n * sizeof(char));
              memcpy(ptr, a, n*sizeof(char));
              nqueens(n, j + 1, a, depth+1);
             free(ptr);

          }
        }
    }
}
```

Figura 4.14 Funció nqueens utilitzant la directiva parallel for

Conclusions

Abans de finalitzar aquest informe comentarem una sèrie de conclusions, obtingudes durant l'estudi de la pràctica. Una conclusió que hem arribat és que, tot i que nosaltres utilitzem les directius correctes de la llibreria OpenMP per paralel·litzar un codi no sempre aconseguirem els resultats més eficients. Aquesta conclusió la obtenim després de fer la segona part del informe, on se'ns demanava paral·lelitzar la funció. Gràcies a les indicacions del professor i al nostre treball ens vam adonar que el codi estava accedint sempre a la mateixa variable, fet que ralentitzava l'execució. Reescrivint dues línies de codi va ser suficient per optimitzar-lo.

És important també el fet de ser conscient que, una funció recursiva paral·lela, si no es controla correctament el seu mecanisme de cut-off, no assolirà bons temps d'execució.