Treball Dirigit

Algorisme de la Bombolla:
Paral·lelització i anàlisi dels resultats

Introducció i Motivació:

El algorisme de la Bombolla (*en anglès "bubble sort"*) és un algorisme d'ordenació molt simple que s'acostuma a ensenyar als estudiants al principi donada la seva facilitat per compendre'l. A nivell pràctic, és un algorisme lent, ja que el seu pitjor cost és quadràtic (i el millor és lineal!), per a grans vectors requereix molt de temps i perd molta eficiència contra algorismes del tipus *quicksort*.

Name	Average	Worst	Memory	Stable	Method
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection

En aquest enllaç podem veure una comparativa entre diferents algorismes d'ordenació.
L'algorisme de la bombolla pot arribar a ser impracticable per a vectors molt grans.

Amb els coneixements obtinguts en aquest curs, la motivació d'aquest treball és comprovar si aplicant tècniques de paral·lelització al algorisme es pot aconseguir millorar la eficiència de l'algorisme, o per contra, la paral·lelització és pitjor que la versió seqüencial.

Per fer tot això possible, primerament analitzarem el codi de l'algorisme de la bombolla i veure quins reptes presenta per a paralel·litzar-lo. Segidament, ens recolçarem en el Tareador per veure les possibles dependències existents i com solucionar-les si cal. Una vegada analitzat passem a implementar la paral·lelització del algorisme amb OpenMP. Finalment, cal fer un anàlisi de si hem aconseguit una millora respecte la versió seqüencial de l'algorisme.

Tots els fitxers i captures es troben disponibles en el repositori de GitHub.

L'algorisme de la bombolla:

Podem resumir el funcionament d'aquest algorisme en recórrer el vector ordenant-lo tantes vegades com calgui fins que tots els seus elements estan ordenats. Es basa en una estratègia de intercanvi de posicions entre la anterior i la següent.

```
void basicbombolla(long n, T data[n]){
  int swaped = 1;
  for(int i=1; i<n && swaped >= 1; i++){
    swaped = 0;
    for(int j=0; j<n-i; j++){
      T dsig = data[j+1], dact = data[j];
      //Sorted less to more
      if(dact > dsig){
        data[j+1] = dact;
        data[j] = dsig;
        swaped++;
      }
    }
  }
}
```

El que fa que aquest algorisme sigui tant lent és precisament tenir que recórrer tot el vector fins que estigui completament ordenat, i en vectors molt grans, aquest temps resulta un problema.

De primeres, no es un algorisme que permeti paral·lelitzar sense fer algunes modificacions.

Si apliquem directament paral·lelització al càlcul del algorisme, obtenim resultats pitjors que seqüencials donat que no es pot processar la següent iteració, dit d'una altre forma, no és pot mirar si el vector està ordenat d'un vector que encara s'està ordenant.

Una primera estratègia que podem aplicar és la de dividir el vector en fragments més petits, i aquests són calculats de manera independent per l'algorisme de la bombolla. Després, només cal anar unint els diferents fragments en un vector resultat. En comptes de paral·lelitzar directament les iteracions del comput de l'algorisme, paral·lelitzem la ordenació de subvectors.

```
void bombolla(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                bombolla(n/4L, &data[0], &tmp[0]);
                bombolla(n/4L, &data[n/4L], &tmp[n/4L]);
                bombolla(n/4L, &data[n/2L], &tmp[n/2L]);
                bombolla(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                basicbombolla(n, data);
        }
}
```

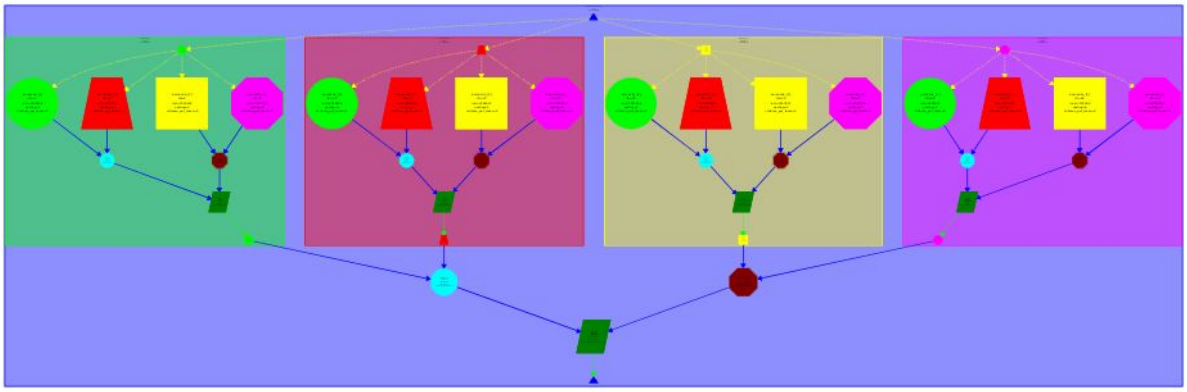
En el codi anterior, es pot apreciar com anem dividint el vector de forma recursiva de 4 en 4 fins a una mida desitjada i apliquem directament l'algorisme de la bombolla. Tot seguit, s'uneixen de forma ordenada els quatre subvectors fins a obtenir l'original però ordenat. D'entrada, aquesta estructura és més fàcilment paral·lelitzable com veurem a continuació amb el Tareador.

Anàlisi amb Tareador

Amb la ajuda del Tareador, modifiquem el codi per veure com podem desenvolupar la paral·lelització. El codi queda de la següent forma:

```
void bombolla(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                  tareador_start_task("bombolla [0]");
                bombolla(n/4L, &data[0], &tmp[0]);
                  tareador_end_task("bombolla [0]");
                  tareador_start_task("bombolla [1]");
                bombolla(n/4L, &data[n/4L], &tmp[n/4L]);
                  tareador_end_task("bombolla [1]");
                  tareador_start_task("bombolla [2]");
                bombolla(n/4L, &data[n/2L], &tmp[n/2L]);
                  tareador_end_task("bombolla [2]");
                  tareador_start_task("bombolla [3]");
                bombolla(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                  tareador_end_task("bombolla [3]");

                  tareador_start_task("merge [0]");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                  tareador_end_task("merge [0]");
                  tareador_start_task("merge [1]");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                  tareador_end_task("merge [1]");

                  tareador_start_task("merge");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                  tareador_end_task("merge");

        } else {
                // Base case
                basicbombolla(n, data);
        }
}
```

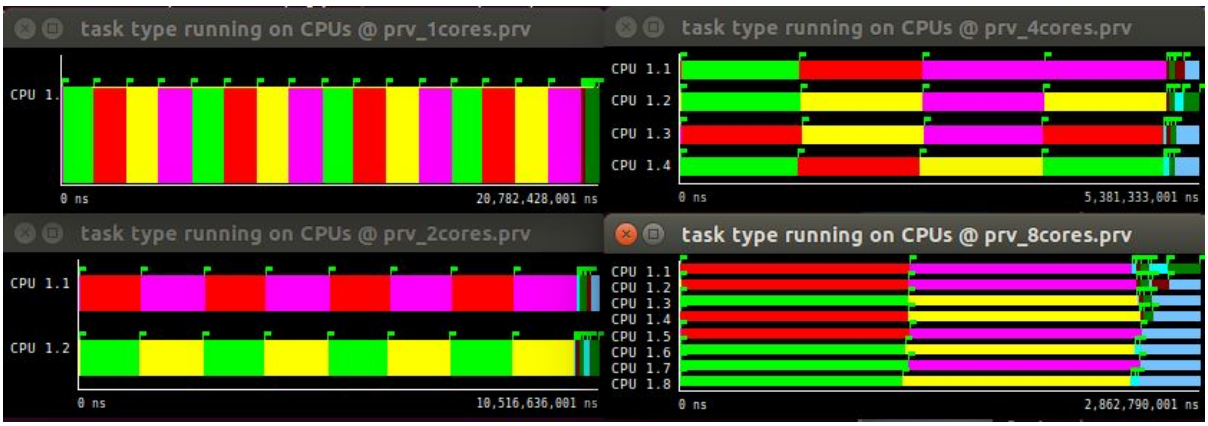
La estratègia consta que cada crida recursiva és una nova tasca a realitzar per un thread. Amb aquesta premissa, comprovem quines són les dependències entre les diferents tasques:



En aquest petit exemple, podem apreciar que el vector original s'ha dividit en 16 subvectors que es poden computar tots a l'hora. Les dependències observables són a l'hora de fer el merge que els 4 subvectors han d'estar completament ordenats.

També podem apreciar la forta escalabilitat que obtenim amb la paralel·lització amb els temps predits pel Tareador:

# Processadors	Temps (s)
1	20,78
2	10,51
4	5,38
8	2,86
16	1,60
32	1,60



Anàlisi de la performance i paral·lelisme amb tasques

Primer, cal crear els threads que s'encarregaran de processar les tasques, i un thread (single) que s'encarregarà de anar-les creant. Això ho fem en la crida a la funció **bombolla(N, data, tmp)** en el main.

```
#pragma omp parallel
#pragma omp single
bombolla(N, data, tmp);
```

Dins de la funció **bombolla**, tenim la part recursiva (que fa 4 crides recursives a si mateixa, dues crides recursives al **merge** i una crida final per unir les dues anteriors) i el cas base, on es fa realment la ordenació. La estratègia *tree* consisteix en crear tasques en les "branques de l'arbre", o dit d'una altre forma, fer una tasca per cada cas recursiu. Per aconseguir això, ens cal una directiva **#pragma omp task** en la crida del **bombolla(...)** i en **merge(...)**.

```
// Recursive decomposition
#pragma omp task
bombolla(n/4L, &data[0], &tmp[0]);
#pragma omp task
bombolla(n/4L, &data[n/4L], &tmp[n/4L]);
#pragma omp task
bombolla(n/4L, &data[n/2L], &tmp[n/2L]);
#pragma omp task
bombolla(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
```

```
#pragma omp task
merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
#pragma omp task
merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

#pragma omp task
merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
```

També ens cal un sincronisme, ja que no podem fer el merge si no hem fet abans l'ordenació. Ens cal tenir totes les parts ordenades per poder unir-les. Per aconseguir això, necessitem esperar a que totes les tasques acabin, i això ho especifiquem amb la directiva **#pragma omp taskwait**.

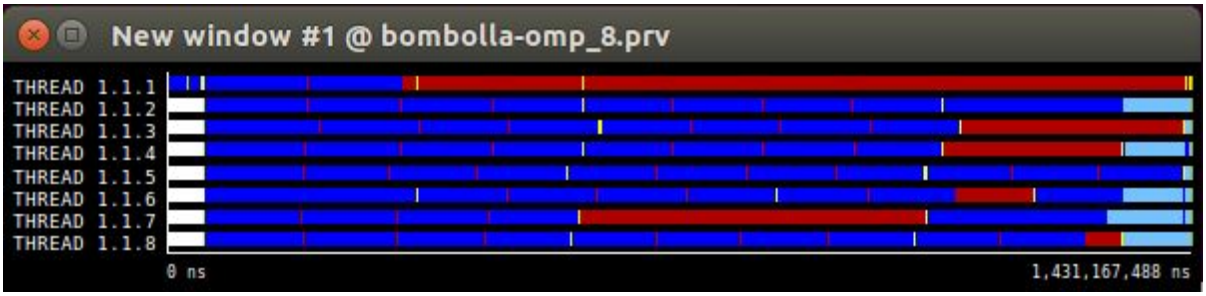
Finalment el codi queda de la següent forma:

```
void bombolla(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task
                bombolla(n/4L, &data[0], &tmp[0]);
                #pragma omp task
                bombolla(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp task
                bombolla(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task
                bombolla(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                #pragma omp taskwait

                #pragma omp task
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                #pragma omp task
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                #pragma omp taskwait

                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                basicbombolla(n, data);
        }
}
```

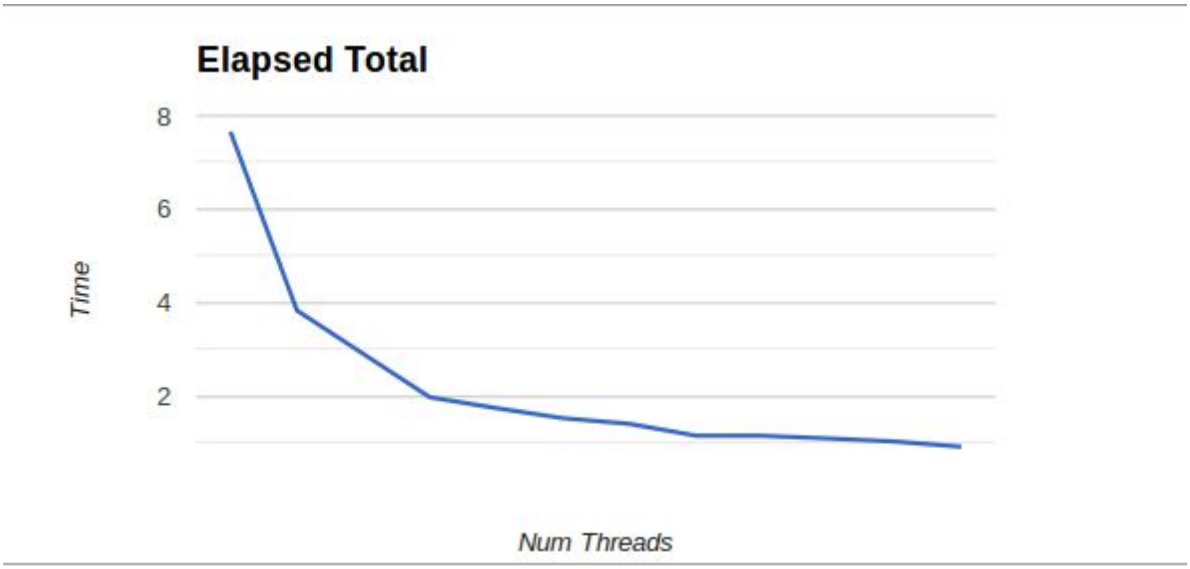
Executem amb 8 threads el codi anterior i obtenim els següents valors:

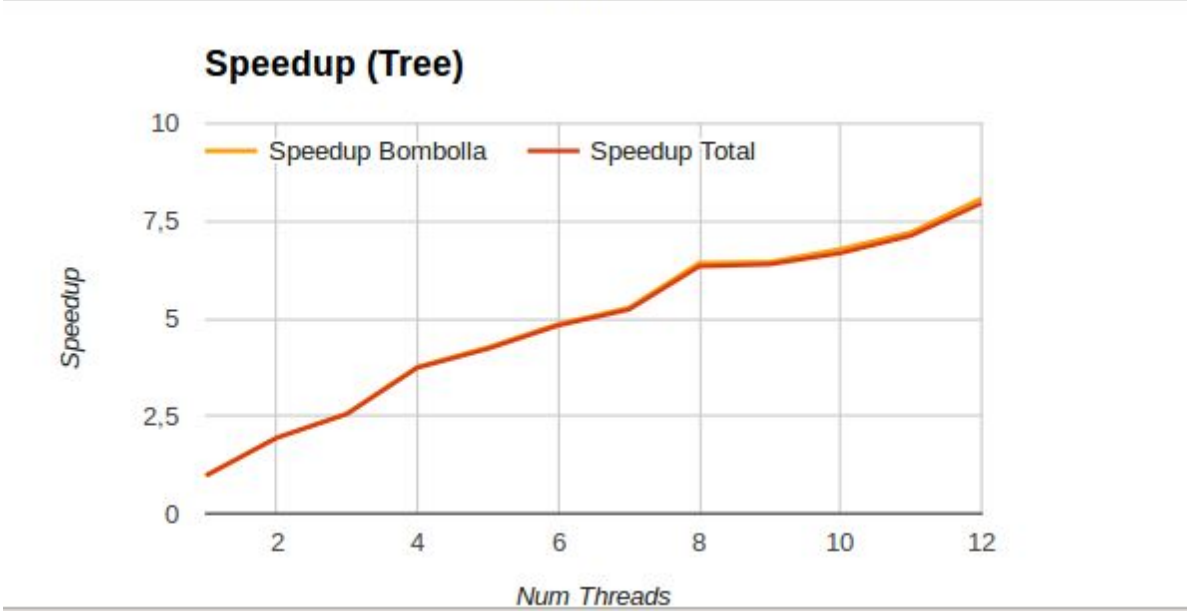
```
Bombolla time in seconds: 0.020661
Bombolla execution time: 1.379860
Check sorted data execution time: 0.000573
Bombolla program finished
```

Amb el paraver podem comprovar que tots els threads treballen al mateix temps:



També fem un estudi de la escalabilitat forta del programa i en especial del algorisme de la bombolla:





Podem apreciar que el cost fort del algorisme és la pròpia ordenació, sense comptar amb la initcialització. Vist els resultats, podem millorar el rendiment del algorisme si en comptes de **taskwait** utilitzem tasques dependents com fem a continuació.

Parel·lització i anàlisis de performance amb tasques dependents

Respecte al codi anterior, els canvis a realitzar són aquests:

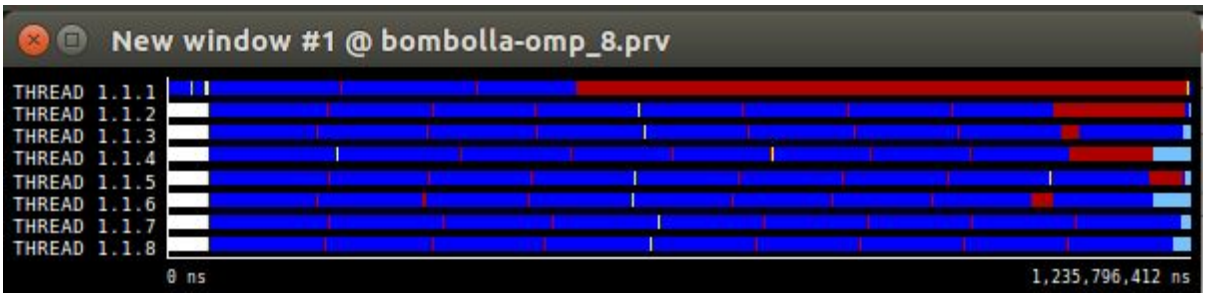
```
void bombolla(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task depend(in: tmp[0]) depend(out: data[0])
                bombolla(n/4L, &data[0], &tmp[0]);
                #pragma omp task depend(in: tmp[n/4L]) depend(out: data[n/4L])
                bombolla(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp task depend(in: tmp[n/2L]) depend(out: data[n/2L])
                bombolla(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task depend(in: tmp[3L*n/4L]) depend(out: data[3L*n/4L])
                bombolla(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                //#pragma omp taskwait //tasques dependents

                #pragma omp task depend(in: data[0], data[n/4L]) depend(out: tmp[0])
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                #pragma omp task depend(in: data[n/2L], data[3L*n/4L]) depend(out: tmp[n/2L])
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                //#pragma omp taskwait //tasques dependents

                #pragma omp task depend(in: tmp[0], tmp[n/2L])
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

                #pragma omp taskwait //Necessari!
        } else {
                // Base case
                basicbombolla(n, data);
        }
}
```

Només cal modificar els "**#pragma omp task**" que teníem definits abans, afegint: "depend(in: *[dades amb la qual te dependència]*) depend(out: *[dades amb la qual generara dependència]*), eliminar els taskwait que ja no serveixen i afegir-ne un al final per assegurar que el merge és correcte.

Executem amb 8 threads per veure si hi ha una millora en el temps:

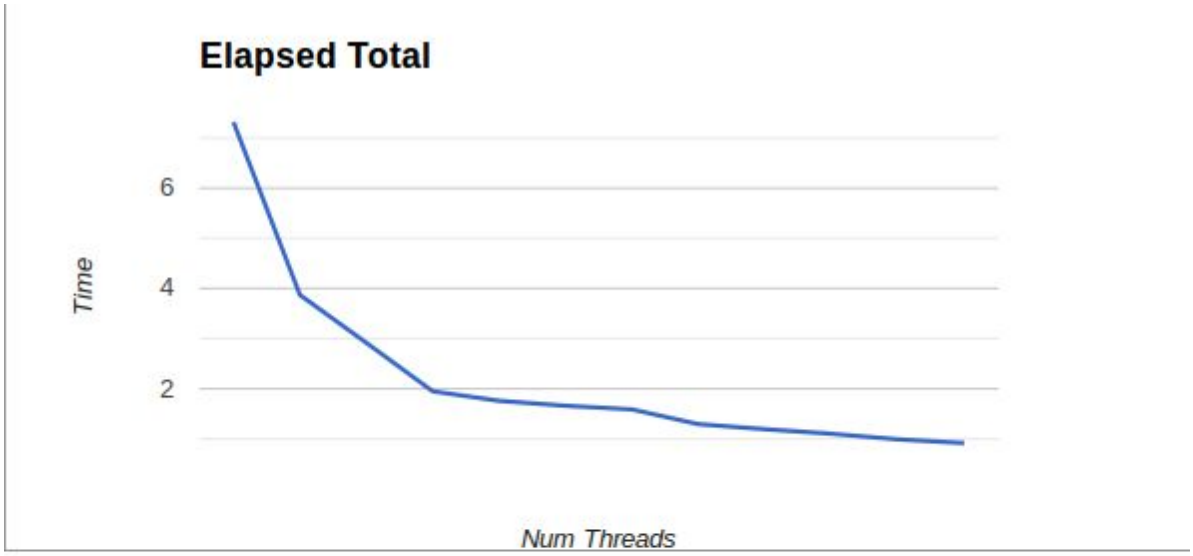
```
Bombolla time in seconds: 0.015048
Bombolla execution time: 1.225994
Check sorted data execution time: 0.000614
Bombolla program finished
```

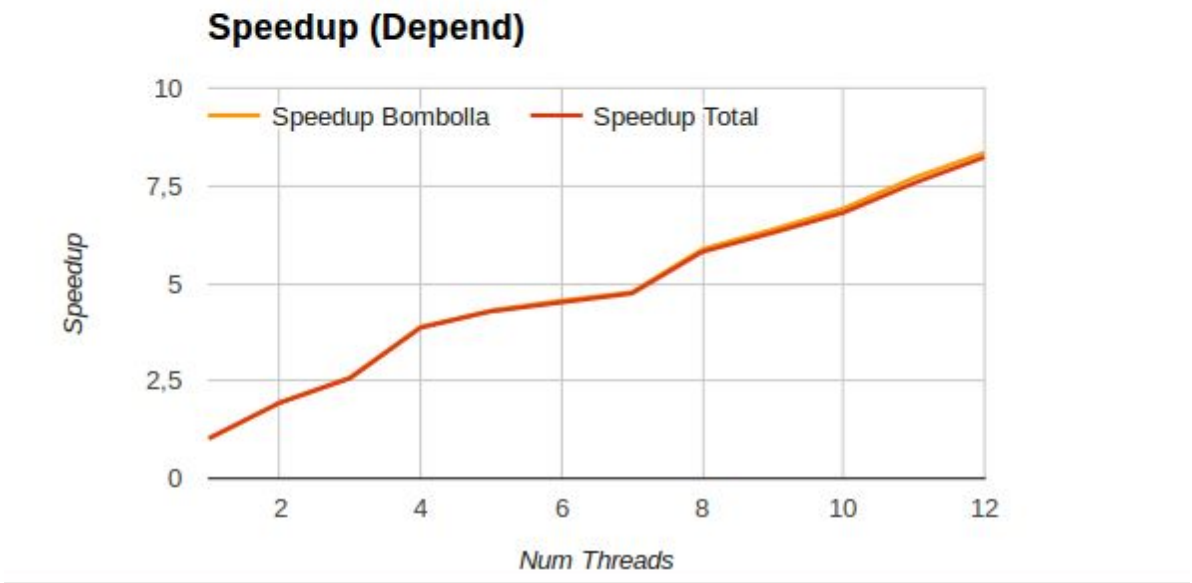
Podem apreciar una petita millora en el temps de l'algorisme. El paraver ens confirma que estem realitzant la parel·lització de forma correcta:



Com podem observar, el cos del algorisme de la bombolla el trobem més compacte. Si anem al detall, la primera diferencia que notem és que el thread 1 ha reduït el tems que estava esperant a la finalització dels altres(zona vermella). Podem apreciar també que els temps d'espera s'han repartit entre tots els threads de forma més eficient.

Això es veu reflectit en la performance de la següent manera:





Conclusions finals:

Després de tot aquest anàlisi, podem extreure les següents conclusions:

- La estratègia de dividir el vector en subvectors, ordenar-los internament i ajuntar-los de forma ordenada permet d'una banda implementar paral·lelisme i per una altra banda millorar els temps (Speedup apox. de 6 punts amb 8 threads).

- L'algorisme de la bombolla és més eficient quan es fa en blocs petits. Però l'augment de blocs comporta perdre més temps en overheads.

- L'algorisme de la bombolla triga massa amb vectors molt grans (tot i que la versió paral·lela és més ràpida, els temps són impracticables).

- No podem fer una paral·lelització més fina donat les limitacions del propi algorisme. És possible de fer-ho d'una altre forma canviant el codi, però no és el objecte d'aquesta practica ni del que es buscava de bon principi.

- Tots els altres elements del codi, ja sigui la initcialització o els merges difereixen en ordres de magnitud del pes de la ordenació mitjançant l'algorisme de la bombolla.

Finalment, apuntar que hi ha algorismes millors com el quicksort, heapsort o mergesort que són més practicables per a vectors grans. El càlcul amb l'algorisme de la bombolla el reservaria per a vectors petits i els elements més o menys ordenats.

Tots els codis utilitzats en aquest treball es troben en el repositori de GitHub personal.
https://github.com/ALEJANDROJ19/Parallel-Bubble-Sort