Cristian Dávila
Enrique González

Deliverables

Deliver a compressed tar file (GZ or ZIP) with the PDF that contains the answers to the questions below and the C source codes with the Tareador instrumentation and with the two OpenMP parallelization versions. In the PDF file clearly state the name of all components of the group and all the necessary information to identify the assignment. Only one file has to be submitted per group through the Raco website.

3.1 Analysis with Tareador

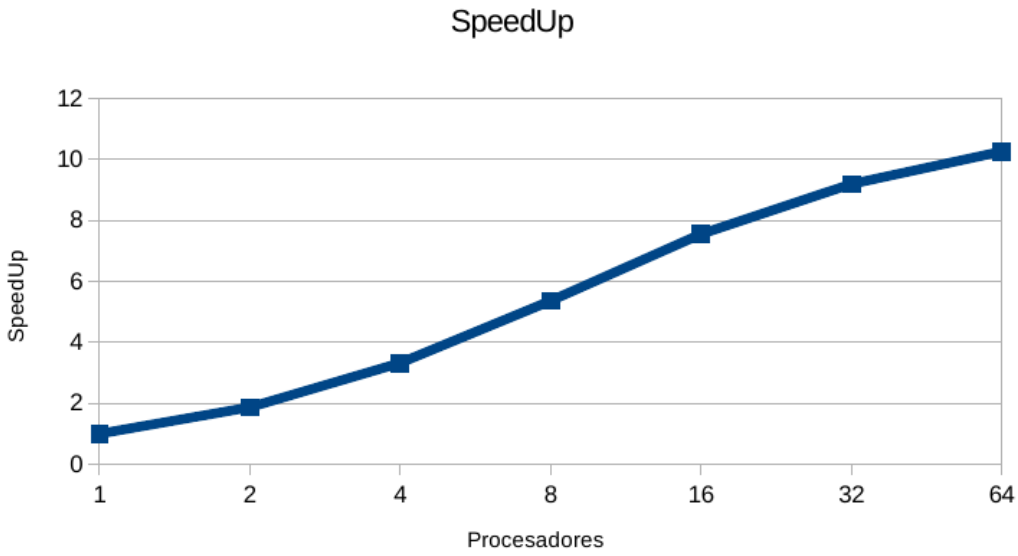
1. Include the source of the sequential multisort.c code modified with the calls to the Tareador API and the task graph generated.

Adjuntamos los códigos y el xdot:

- multisort.c
- multisort-omp-leaf.c
- multisort-omp-tree.c
- dep_graph_multisort-tareador.xdot

2. Write a table with the execution time and speed-up predicted by Dimemas (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

# CPUs	Estimated time	Speed Up
1	50,432,829 µs	1
2	26,968,544 µs	1.87
4	15,183,264 µs	3.32
8	9,402,834 µs	5.36
16	6,682,881 µs	7.55
32	5,487,019 µs	9.19
64	4,917,609 µs	10.26



Podemos ver en los resultados obtenidos con Dimemas que el tiempo de ejecución y el *Speed Up* mejoran con el aumento de procesadores, aunque no llegamos a obtener los resultados de un caso ideal. Ya que en un supuesto caso ideal el *Speed Up* debería ser igual al número de procesadores.

Con pocas *CPUs* vemos como no se aleja demasiado, pero con el aumento de los procesadores cada vez nos alejamos más del caso ideal en el que, por ejemplo, al trabajar con 64 procesadores deberíamos obtener un *Speed Up* de 64, frente al 10.26 que nosotros hemos obtenido.

3.2 Parallelization with OpenMP

3. Briefly describe the two versions (Leaf and Tree) implemented, making references to the source code included in the compressed tar file.

En la versión *Leaf* las llamadas recursivas las ejecuta un único thread secuencialmente, y los casos base (hojas) serán los que se ejecuten en tareas en paralelo. Tan solo mirando el código también podemos ver que en esta versión como máximo tendremos 4 tareas ejecutándose en paralelo, las 4 llamadas al *multisort* que tratarán el caso base, llamando al *basicsort.*
En cambio en la versión *Tree* las llamadas recursivas se ejecutarán en paralelo, por lo tanto en este caso el número máximo de tareas que podemos llegar a tener ejecutándose en paralelo es igual al número de threads.
Por lo tanto, podremos ver que tanto el tiempo de ejecución como el *Speed Up* de las dos versiones será similar hasta llegar a los 4 threads. Una vez pasemos a trabajar con más de 4 threads la versión *Tree* seguirá mejorando, mientras que la versión *Leaf* no.

3.3 Performance analysis

Write a text, inspired in the one provided below, summarizing the performance results for the two versions (Leaf and Tree) of multisort.

The performance of the two parallelization strategies for multisort has been analyzed on a multiprocessor architecture with 12 Intel cores with the following results:

Leaf Scalability with 8 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	1.387980	1	0.399170	1	0.489154	1
1	1.394420	0.9953	0.398815	1.0008	0.483787	1.0110
2	0.753335	1.8424	0.214427	1.8616	0.270303	1.8097
4	0.374492	3.7063	0.119721	3.3342	0.153398	3.1888
6	0.369695	3.7544	0.123649	3.2283	0.149013	3.2826
8	0.361441	3.8401	0.124612	3.2033	0.150722	3.2454
10	0.368220	3.7694	0.123355	3.2359	0.149526	3.2714
12	0.369639	3.7549	0.117053	3.4102	0.153729	3.1819

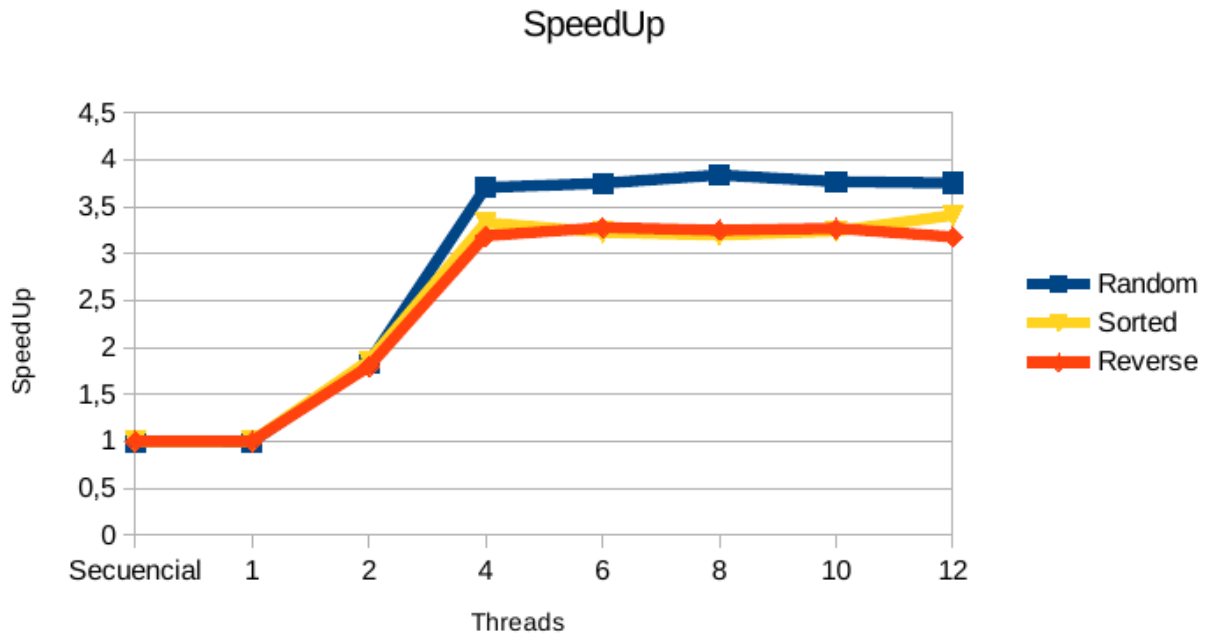


Figure 2: Plot Speed Up of the Leaf Scalability with 8 Megaelements size

Tree Scalability with 8 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	1.392510	1	0.401878	1	0.488697	1
1	1.387430	1.0037	0.398857	1.0076	0.485485	1.0066
2	0.717853	1.9398	0.211934	1.8962	0.265499	1.8407
4	0.371508	3.7483	0.114899	3.4977	0.137607	3.5514
6	0.349378	3.9857	0.092277	4.3551	0.127751	3.8254
8	0.285531	4.8769	0.087088	4.6146	0.104605	4.6718
10	0.232558	5.9878	0.071385	5.6297	0.082851	5.8985
12	0.206309	6.7496	0.072794	5.5208	0.083454	5.8559

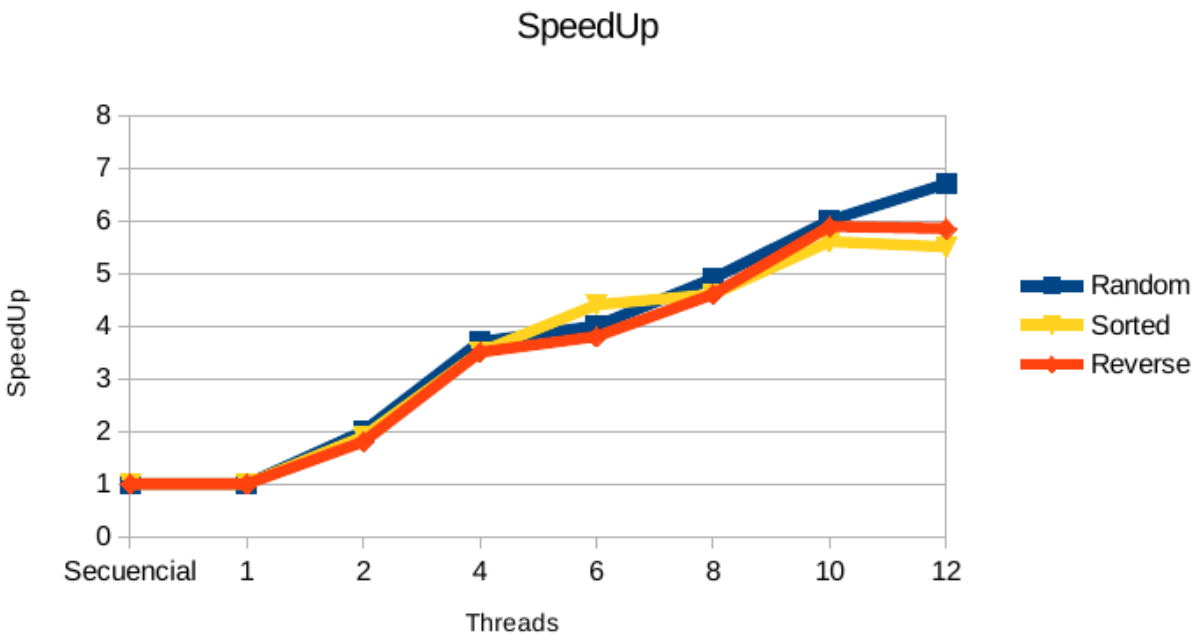


Figure 2: Plot Speed Up of the Tree Scalability with 8 Megaelements size

For all the performance results we have used an input vector of size 8192 Kiloelements randomly generated by the program itself. The program performs several sort steps in order to verify the influence of the randomness of the data in the input vector. The recursive depth of the multisort algorithm is controlled with the value of the global variable MIN_SORT_SIZE, when after a few recursive calls the lenght of the vector is less to this value we will carry out the basic case. The first conclusion of the analysis is that version Tree has better scalability than version Leaf.

This is due to the fact that the parallelism exploited in version Leaf is limited by the maximum number of tasks we can run in parallel, which are 4 because when we achieve the maximum depth we will make 4 calls to multisort to treat the base case. Then we'll wait for synchronize the 4 tasks, this synchronization is the cause of the bound because the main thread, which is executing all the recursive calls, is waiting for the 4 sorted vectors of the base cases.
Figure 1 shows the execution timelines visualized with Paraver, supporting the previous argument.

Figure 1: Timelines visualized with Paraver.

Figure 2 plots the speedup, with respect to the sequential, for different vector sizes (8 Megaelements, plots viewed above; 16 and 32 Megaelements below), for the two OpenMP versions and for the different invocations of multisort in the main program.

Leaf Scalability with 16 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	2.894700	1	0.831319	1	1.078570	1
1	2.902320	0.9974	0.829062	1	1.078920	0.9996
2	1.534520	1.8863	0.453681	1.8323	0.590068	1.8278
4	0.883490	3.2764	0.337115	2.4659	0.393590	2.7403
6	0.764722	3.7852	0.238433	3.4865	0.319972	3.3708
8	0.717804	4.0327	0.241597	3.4409	0.318131	3.3903
10	0.731089	3.9594	0.247199	3.3629	0.326019	3.3083
12	0.731862	3.9552	0.245923	3.3804	0.316979	3.4026

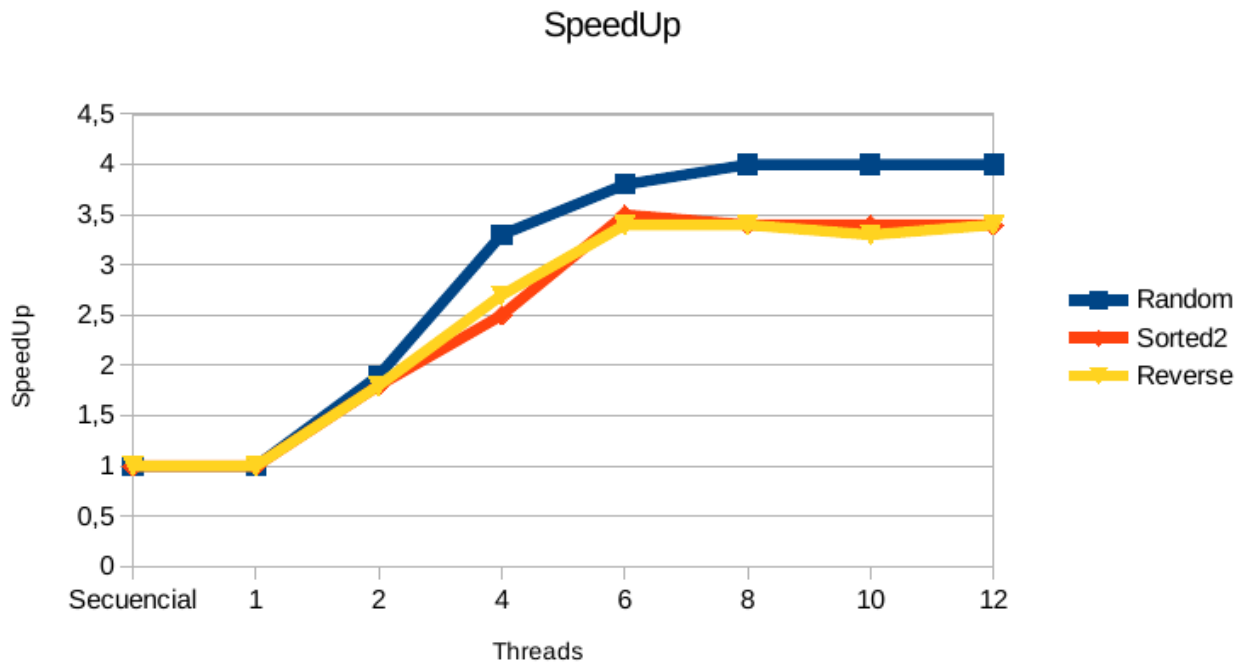


Figure 2: Plot Speed Up of the Leaf Scalability with 16 Megaelements size

Leaf Scalability with 32 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	5.947710	1	1.704710	1	2.183110	1
1	5.991050	0.9927	1.702700	1	2.172220	1.0050
2	3.279650	1.8135	0.933992	1.8251	1.207040	1.8086
4	1.585880	3.7504	0.500207	3.4080	0.637007	3.4271
6	1.525510	3.8988	0.513692	3.3185	0.629023	3.4706
8	1.495110	3.9781	0.515665	3.0584	0.636451	3.4301
10	1.455950	4.0851	0.519587	3.2808	0.641291	3.4040
12	1.519040	3.9154	0.541790	3.1464	0.678589	3.2171

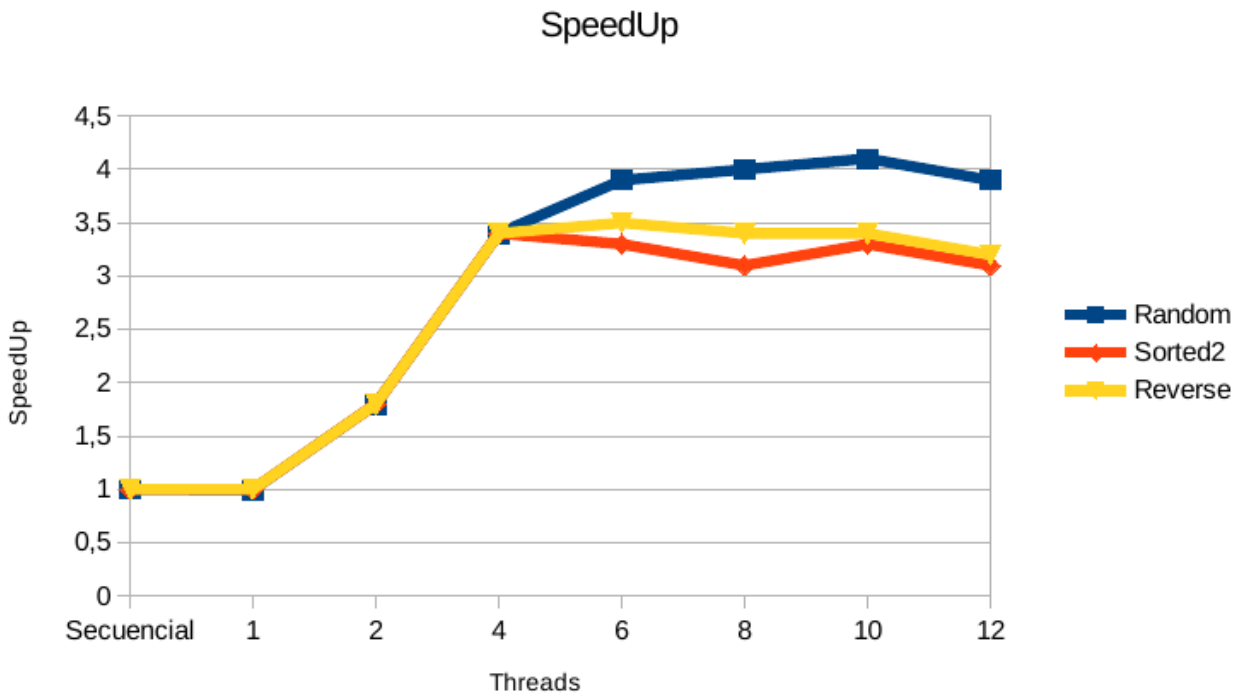


Figure 2: Plot Speed Up of the Leaf Scalability with 32 Megaelements size

Tree Scalability with 16 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	2.892300	1	0.831462	1	1.075130	1
1	2.894830	0.9991	0.845354	0.9835	1.072150	1
2	1.531630	1.8883	0.458497	1.8134	0.601910	1.7861
4	0.794121	3.6421	0.240450	3.4579	0.316566	3.3962
6	0.718465	4.0256	0.227455	3.6555	0.238343	4.5108
8	0.566481	5.1057	0.178768	4.6510	0.226060	4.7559
10	0.451325	6.4084	0.142592	5.8310	0.182889	5.8785
12	0.407244	7.1021	0.140864	5.9025	0.180028	5.9720

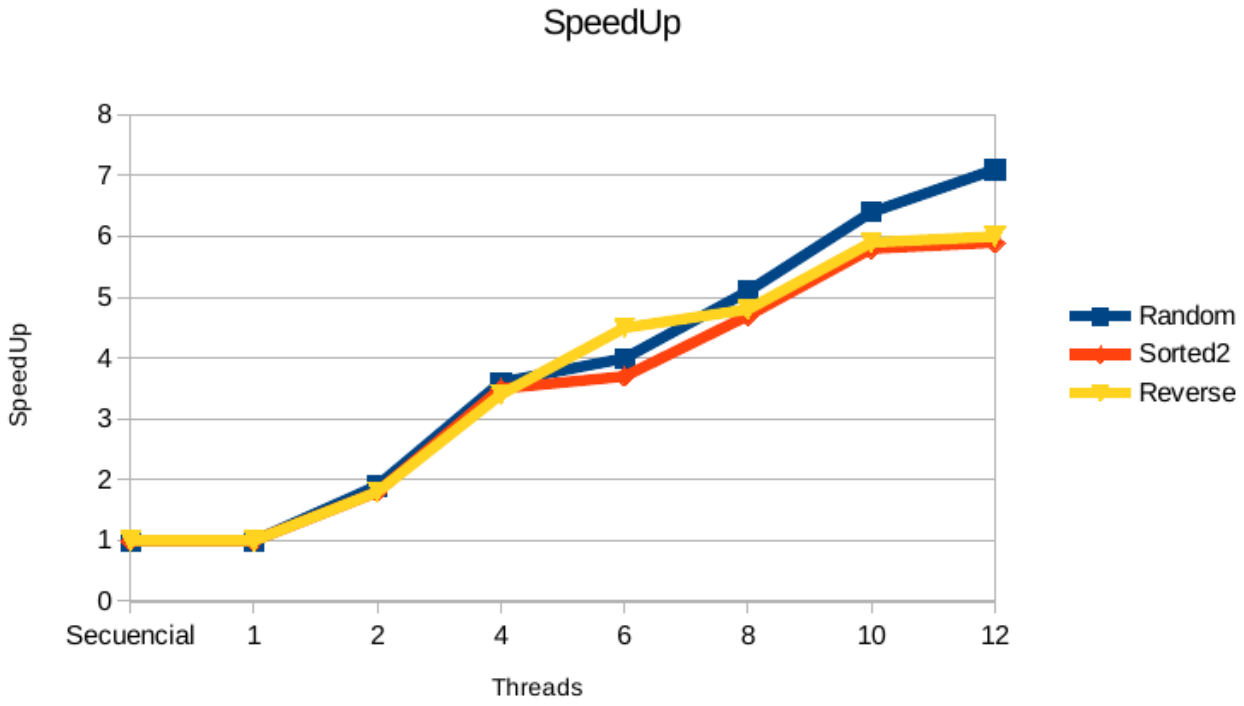


Figure 2: Plot Speed Up of the Tree Scalability with 16 Megaelements size

Tree Scalability with 32 Megaelements size:

	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpRe
Secuencial	5.959540	1	1.703940	1	2.158450	1
1	5.949520	1	1.703330	1	2.186910	0.9869
2	3.131040	1.9033	0.939243	1.8141	1.157210	1.8652
4	1.593820	3.7391	0.495253	3.4405	0.611511	3.5296
6	1.342120	4.4403	0.401730	4.2415	0.493728	4.3717
8	0.997870	5.9722	0.352254	4.8372	0.435653	4.9545
10	0.868113	6.8649	0.292732	5.8208	0.359096	6.0107
12	0.743877	8.0114	0.264409	6.4443	0.329087	6.5589

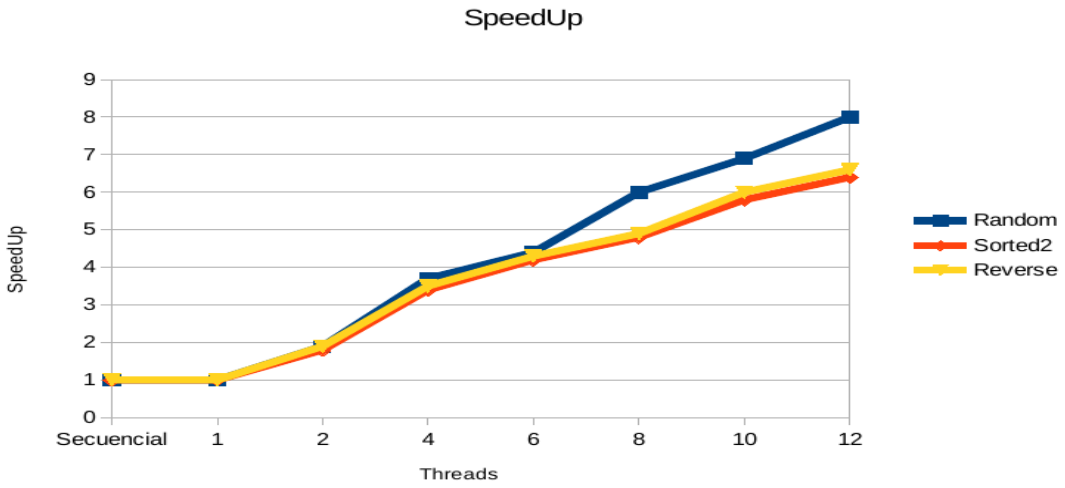


Figure 2: Plot Speed Up of the Tree Scalability with 32 Megaelements size

We see at these tables and plots what we have explained before, the Leaf version does not improve from the 4 threads because of the limitation with the maximum number of tasks that will can do in parallel. However we have seen that the Tree version still improving with more than 4 threads, as it haven't any limitation like the Leaf version.
Finally, we have also analyzed the influence of the recursively depth in the tree version. Figure 3 plots the speedup when changing the recursion depth, for a vector of 16 Megaelements and 8 threads. We observe an optimal point around the 8 Kiloelements value. In fact, we observe that for a sequence of consecutive values, between 8 and 128 Kiloelements we obtain almost the same results. Therefore is pretty hard to give a single value.

OpenMP execution with 8 threads

Depth (Kelements)	Random	SpeedUpR	Sorted	SpeedUpS	Reverse	SpeedUpR
1	0.564251	1	0.550643	1	0.531293	1
2	0.503045	1.1216	0.217200	2.5351	0.240842	2.2059
4	0.508425	1.1098	0.176622	3.1176	0.206593	2.5716
8	0.412295	1.3685	0.160286	3.4353	0.191741	2.7700
16	0.473629	1.1913	0.164818	3.3409	0.195027	2.7242
32	0.471617	1.1964	0.167012	3.2970	0.190849	2.7838
64	0.478813	1.1784	0.163770	3.3622	0.195475	2.7179
128	0.478971	1.1780	0.167719	3.2831	0.192010	2.7670
256	0.529365	1.0659	0.162174	3.3953	0.191526	2.7739
512	0.573949	0.9831	0.180121	3.0570	0.228207	2.3281
1024	0.574905	0.9814	0.179464	3.0682	0.227376	2.3366
2048	0.769671	0.7331	0.235270	2.3404	0.330019	1.6098
4096	0.800560	0.7048	0.240816	2.2865	0.312115	1.7022
8192	3.041500	0.1855	0.874888	0.6293	1.173410	0.4528

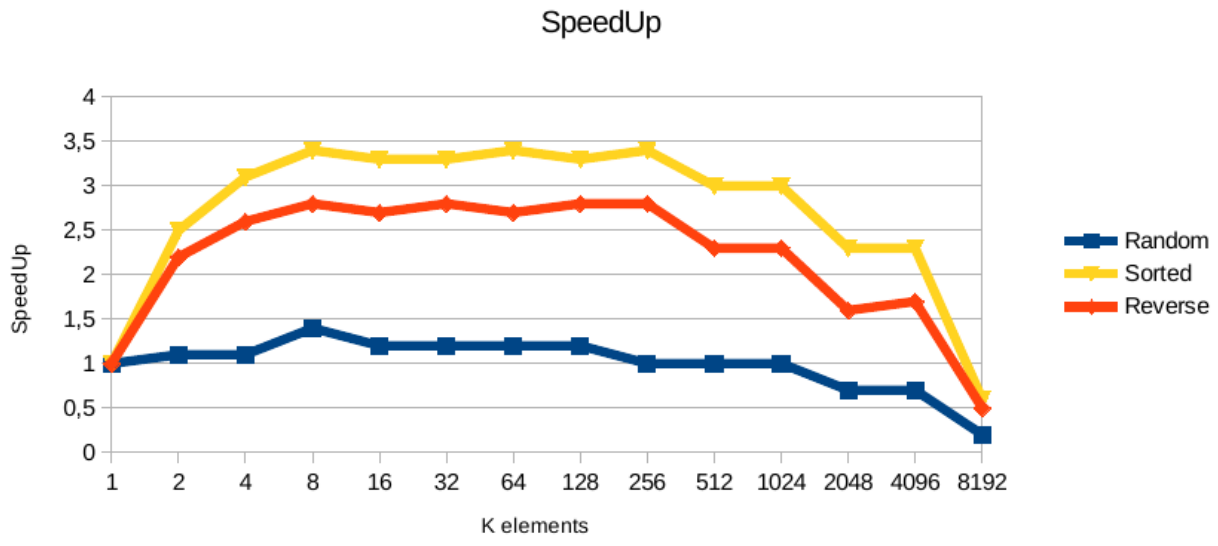


Figure 3