CPA Lab 2

Carlos Santiago Galindo Jiménez
Jesús Vélez Palacios

November 7, 2016

1 Objective

Parallelize the execution of a program that reconstructs an image whose rows have been shuffled. Study different approaches and some improvements to the algorithm.

Examine the temporal cost for each version for different amounts of threads. Detail also the speedup and efficiency.

2 Exercises[1]

`encaja-e1.c` Time the method encaja. Completed with two calls to `omp_get_wtime()`.

`encaja-e2-p?.c` Parallelize the different loops (if at all possible).

- `i` This loop will never be parallelizable, because each line depends on the previous one.
- `j` This loop can be parallelized protecting the variables with `private(x, distancia)`. The last `if` must be protected as `critical section` and the condition has to be rechecked after entering it. Source code
- `x` This loop can also be parallelized, and needs a `reduction(+:distancia)` to compute the distance correctly. Source code

`encaja-e3.c` Improve the program by exiting loop `x` if the partial sum surpasses the current minimum. Solved adding an `if` that breaks from the loop in that case.

`encaja-e4-p?.c` Parallelize `encaja-e3.c` in the loops that are viable.

- `j` This loop is parallelized in the same way as in exercise 2. Source code
- `x` This loop needs the most work. It must be parallelized as a `parallel` block. The first iteration is assigned to `omp_get_thread_num()` and each thread increments `x` by `omp_get_num_threads()`. It still needs a `reduction` for `distancia`. Source code

3 Performance

3.1 Expected results

Exercise 2: encaja-e2

In general, the outer loops will achieve better parallel performance than inner loops. This is due to the overhead of creating a thread, meaning the extra resources the system spends creating and destroying the threads. There is a point where the overhead becomes so great that there is no speedup. For this reason it is expected that `encaja-e2-pJ.c` (n threads) will perform better than `encaja-e2-pX.c` (n^2 threads).

wording here?

Exercise 3: encaja-e3

The change proposed should mean an improvement in time required to recover the image.

[1] All files are provided in the appropriate task. However, the files are also hosted on GitHub. These files will be exactly equal to those sent. Some references even link to a certain line in the file, for greater clarity.

Exercise 4: encaja-e4

In this case, the same should happen as in exercise 2: `encaja-e4-pJ` should run faster than `encaja-e4-pX`. It is more difficult to compare each with the rest of iterations.

3.2 Testing

All the programs have been run and timed in the Kahan Cluster, from the university. The job submission file for that system is provided along with the source code as `encaja.sh`. This file executes all the exercises, and prints the results in `csv` format (values separated by commas) for easier data processing. A `Makefile` is also included with the source code, because `encaja.sh` makes use of it.

3.3 Results: time required

The sequential programs (`encaja-e1.c` and `encaja-e3.c`) have only been tested once. The parallel programs have been run with 2, 4, 8, 16 and 32 threads to test their performance. As shown in the graph[2] from Figure 3, loop `j` achieves much better results that loop `x`. On top of that, the improvement implemented in the exercises 3 and 4 pays off in the sequential version and loop `j`, but in loop `x` there is practically no difference with `encaja-e2-pX`. In the latter case, the overhead of the threads gets so big that the algorithm gets slower.

Program	Execution time
`encaja-e1`	15.257217
`encaja-e3`	2.725587

Figure 1: Sequential execution times

Program	2t	4t	8t	16t	32t
`encaja-e2-pJ`	6.938746	3.548883	1.970450	1.183996	0.726187
`encaja-e4-pJ`	1.439842	0.795035	0.496000	0.330912	0.256960
`encaja-e2-pX`	8.337415	5.464151	5.557857	8.197272	12.789012
`encaja-e4-pX`	4.065065	4.217589	5.164125	8.656438	13.590685

Figure 2: Parallel execution times

3.4 Results: speedup

The speedup of a parallel program respect to the sequential version is the improvement in speed due to the parallelization. It is computed as:

$$S(n,p) = \frac{t(n)}{t(n,p)} \tag{1}$$

Applying the previous formula with $t(n) = t_{encaja-e1}$ and $t(p,n)$ equal to the table in Figure 2 we obtain Figure 4. Values $S(n) > 1$ indicate an improvement and values $S(n) < 1$ signal a worsening in the performance. All values are > 1, therefore there is improvement in all cases. The highest result is obtained on `encaja-e4-pJ`, with a 5830% speedup.

The speedup can also be computed with $t(n) = t_{encaja-e3}$, in order to compare exercise 4 its sequential version (activity 3).

In this table it can be appreciated that the loop `x` has all values $S(n,p) < 0$, and therefore provides no improvement the sequential code in `encaja-e3.c`. The apparent betterment shown in Figure 4 is respect to the initial program. The threads — time graph in Figure 3 clearly shows that the performance of `encaja-e4-pX.c` (yellow) is below all other versions.

[2]The data in the graph for *nodes* = 1 is the timing from the sequential versions of the program (`encaja-e1.c` and `encaja-e3.c`)

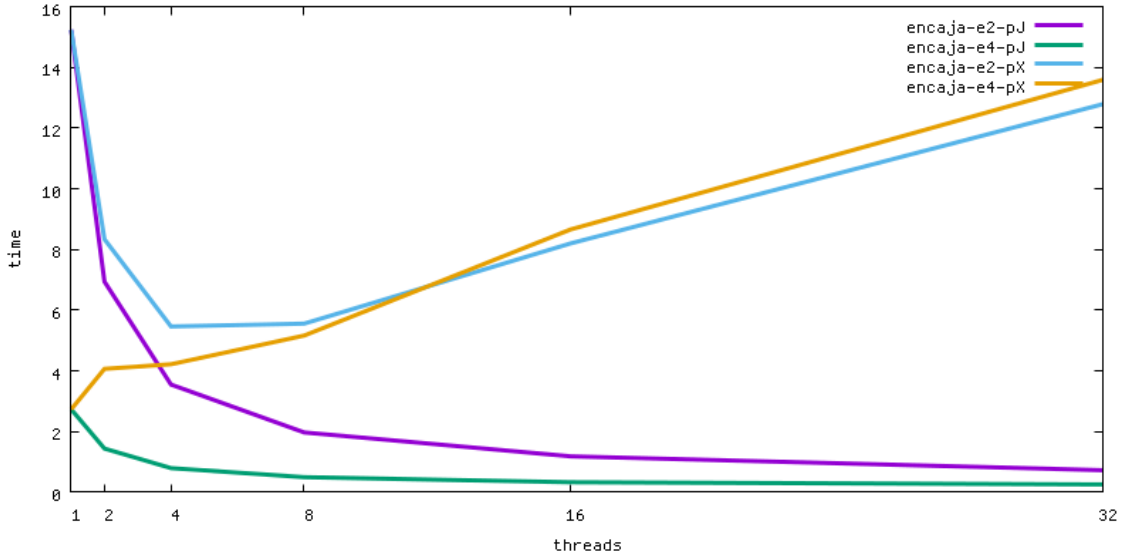

Figure 3: Threads — time graph

Program	2t	4t	8t	16t	32t
`encaja-e2-pJ`	2.19624	4.29407	7.73384	12.87090	20.98510
`encaja-e4-pJ`	10.58390	19.16790	30.72410	46.05190	59.30550
`encaja-e2-pX`	1.82780	2.78893	2.74191	1.85905	1.19158
`encaja-e4-pX`	3.74881	3.61323	2.95096	1.76044	1.12129

Figure 4: Speedup

3.5 Results: efficiency

The efficiency of a parallel program is defined as:

$$E(n,p) = \frac{S(n,p)}{p} \tag{2}$$

where p is the number of threads and $S(n,p)$ the speedup previously computed. Figure 7 shows the efficiency values for exercises 2 and 4. This metric means

complete this

4 Conclusions

This lab practice is an extensive walk-through of OpenMP and parallelization practices and techniques. In this weeks we have learned that:

- Some loops can be parallelized, some cannot.
- Thread overhead may become a problem with inner loops.
- Clever changes to sequential algorithms can outperform simpler parallel ones.
- Appropriate control structures for critical sections and shared, private variables.

write conclusions

Program	2t	4t	8t	16t	32t
`encaja-e4-pJ`	1.898620	3.438490	5.511530	8.261170	10.638700
`encaja-e4-pX`	0.672491	0.648171	0.529368	0.315802	0.201147

Figure 5: Speedup of `encaja-e4` respect to `encaja-e3`

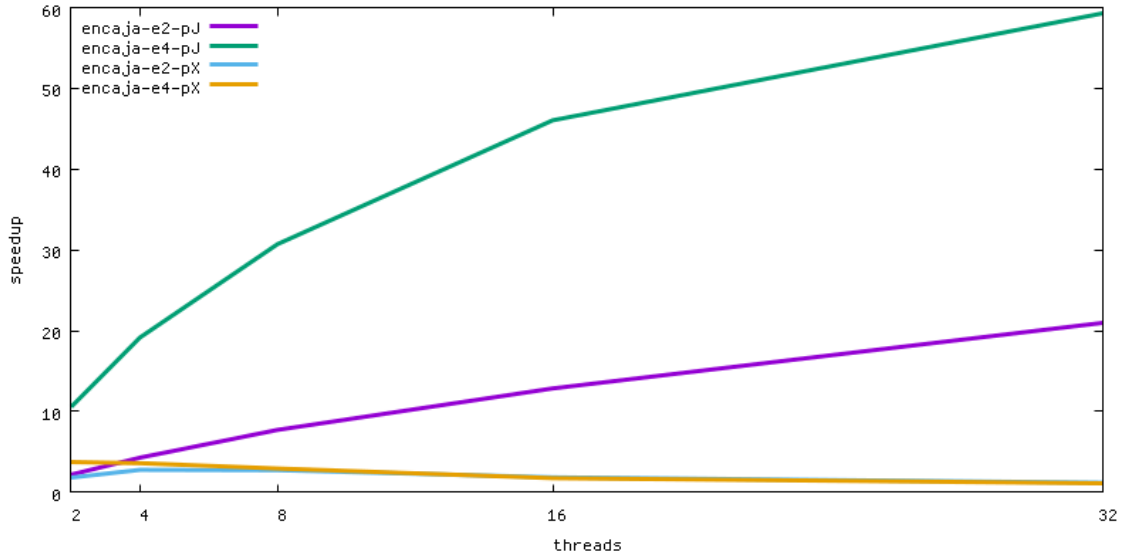

Figure 6: Threads — speedup graph

Program	2t	4t	8t	16t	32t
`encaja-e2-pJ`	1.09812	1.073520	0.966730	0.804431	0.655784
`encaja-e4-pJ`	5.29195	4.791970	3.840510	2.878240	1.853300
`encaja-e2-pX`	0.91390	0.697233	0.342739	0.116191	0.037237
`encaja-e4-pX`	1.87441	0.903308	0.368870	0.110028	0.035040

Figure 7: Efficiency

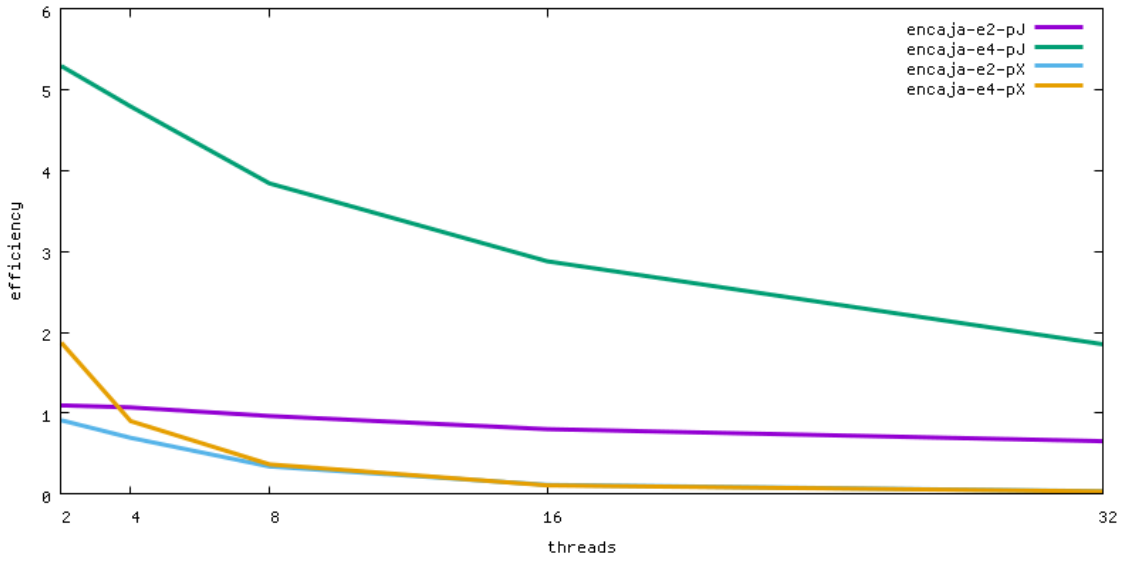

Figure 8: Threads — efficiency graph