1 Heat transfer

1.1 Assumptions

We don't assume anything more than what explained in the assignment :

- We have a square, of side length even and ≥ 2,
- The 4 centre cases (the *heat core*) have an unknown but identical initial value ; this must and will remain unchanged throughout the iterations,
- The border cases(the *heat sink*) must and will always stay null,
- All the cases other than the cores are initialized to 0.

All these points allow us to assume a general symmetry, along 4 axis : vertical, horizontal and both diagonals. This will be useful later on.

1.2 Measurements

1.2.1 Multithreading

First of all, we add multithreading. We could naively try to parallelize iterations, but as each iteration completely depends on the previous one, this can't be done. Our next reasoning is to parallelize according to rows. This way, each thread will run through multiple rows and through all columns (of its rows). Note that thanks to the construct `#pragma parallel for`, the work of the `for` loop is automatically balanced between threads. A more complicated version could have used half of the threads to iterate through rows, and each thread would spawn 2 threads to iterate the columns.

1.2.2 Switching

Method	**value**	**1 threads**	**2 threads**	**4 threads**	**8 threads**	**16 threads**
No switching	[s]	184.90	123.50	94.46	82.9	74.46
	speedup	-	*1.49*	*1.95*	*2.35*	*2.48*
Switching	[s]	161.8	84.67	45.91	27.68	18.63
	speedup	-	*1.91*	*3.52*	*5.84*	*8.68*
Speedup $^{with}/_{without}$		1.14	1.46	2.06	2.99	3.79

Table 1: Whole square with and without switching

We start off by a simple and naïve algorithm : We iterate through the whole table, read values from `input`, compute their mean, and write it to `output`. At the end of the iteration (when all threads have joined), we copy all values from `output` to `input`[1]. Then, we add a first improvement : instead of doing a `memcpy` at each iteration, we do a **switching**.

[1] If we needed to keep `input` intact, we could initialize a temporary table and use it as a substitute for `input`. The result would be very similar, as it only requires an additional `malloc` and `memcpy`.

Method	**Value**	**1 threads**	**2 threads**	**4 threads**	**8 threads**	**16 threads**
Half horizontal	[s]	104	54.16	31.49	19.57	15.28
	speedup	-	*1.92*	*3.30*	*5.31*	*6.81*
Half vertical	[s]	112	58.72	33.18	18.27	13.15
	speedup	-	*1.91*	*3.38*	*6.13*	*8.52*
Quarter	[s]	185.3	123.43	95.54	80.09	75.57
	speedup	-	*1.50*	*1.94*	*2.31*	*2.45*
Eighth	[s]	457.3	338.2	198.16	106.76	59.04
	speedup	-	*1.35*	*2.31*	*4.28*	*7.75*

Table 2: Comparison of more clever algorithms

Depending on the iteration, we will read `input` and write to `output` or the opposite. After all iterations, if necessary, we adjust by a single `memcpy`. We can see in table 1 the results :

Little note on the table : each method is presented with its speedup, that is the fraction between the 1-threaded result and x-threaded. The last row of the table presents the speedup between each with/without switching, for each amount of threads.

The effect of the *switching* is clear : the `memcpy` is clearly the heaviest part of the computation and thus removing it creates a huge speedup (between 1.14 and 3.79). Note also that without the *switching*, the speedup induced by multi-threading is not great (as 16 threads only induce a 2.5 speedup). On the other hand, the part with *switching* benefices from a huge speedup (up to 8.68) thanks to the multi-threading.

1.2.3 Separating the square

Now we know using switching is a good idea. We then try a new improvement : not visiting the whole table, but only parts. As the square is... square, it's perfectly symmetrical. This means we only have $^1/_8$ of interesting data, the rest is just redundant. The goal is to compute the mean for only a fraction of the square, and then do multiple assignments (with 1 mean, fill multiple cases). We compare different slicing : half (vertically and horizontally), quarter and eighth. All results and speedups (compared to 1-threaded version of the same algorithm) are visible in table 2.

But clearly there is a problem : doing $^1/_8$ of the work results in significantly longer time. Indeed, there is a huge speedup (factor 8 for 16 threads), but the time is even longer than the *Whole with switching*. This deserves some explanation : indeed, we are doing less reading but as much writing. And we need to write at so many different locations (8 for the last algorithm), the cache has to discard a lot of data. When working with *half* (e.g. horizontal, the most efficient), a thread "owns" certain lines and the other threads will never touch it. And the pre-fetching is almost optimal, as we will load as much as possible, and the data won't be "useless"[2]

[2] Because the pre-fetched data will be read almost always, when in other algorithms some pre-fetched data will be discarded without having been read.

1.2.4 Improving cache usage and delaying writings

So the best algorithm seems to be cutting the table in half. But still, there is a lot of redundancy not exploited. The *eighth* version is not good, because of the reasons discussed above. Fetching (0,0) will pre-fetch (0,1), (0,2),... but none of these will be read. And we still have a lot of writing. But what if we could only fetch useful data and delay writings ? This is the goal of the algorithm **simulate** : we rearrange the data in a new placeholder that is significantly smaller (about $^1/_8$ of the cases of the original). And with this rearranged data model, 2 improvements are made : we are exploiting space locality and reducing number of writings.

Space locality is used, because with a good fraction of the table, we can pre-fetch useful data. Indeed, now $(1, 1)$ is next to $(2, 0)$ and $(2, 1)$. And we need those to compute the mean at $(1, 1)$ while we surely don't need $(1, 32)$ which was pre-fetched before. The function `meanify` ensures the correctness of the computation. For example, when computing the mean of a diagonal element, we should use data at a position non-existent in our new memory model. But this non-existent data can can be inferred using symmetry. We therefore extract several cases :

- The diagonal : 3 elements don't "exist", so we use symmetry on those. They are elements of the form (i, i).
- Special case of the diagonal : the lase case, being in the centre square. We don't modify this one.
- "Below diagonal" : the cases below the diagonal elements, with the form $(i, i - 1)$. Only 1 element can't be found, and inferred using symmetry.
- The element "below diagonal" of the last row is a special case, with 3 inexistent elements.
- The last row : elements of the form $(length/2, j)$ are out of the memory, but with the same value as $(lenght/2 - 1, j)$, that represent our last row. We use symmetry on those.

Even if this improves the use of the cache, this is still not perfect. Indeed, there are still rows where we can't pre-fetch 3 rows at the same time, as they grow too big. But this is still better than the naive model.

Writings reduction is made by only considering this eighth. Contrary to the previous *eighth* algorithm, we don't make 8 writings per mean calculation. We only make 1, and at the very last iteration, we write to the output table. So only this iteration is longer.

Measurements are presented in table 3. Note that the last line presents the speedup to the 1-threaded naive `whole no-switching`. We see we obtain a 42.31 speedup with our best result.

Method	**Value**	**1 threads**	**2 threads**	**4 threads**	**8 threads**	**16 threads**
Simulate	[s]	28.38	21.48	13.09	7.442	4.37
	speedup	-	*1.32*	*2.17*	*3.81*	*6.49*
Speedup to naive		6.52	8.61	14.13	24.85	42.31

Table 3: Execution time of the `simulate` method

1.3 Going further

The slicing and memory rearrangement was only possible with what we know about the image (especially that it's 4-way symmetrical). If we could not make this assumption any more, we would implement different algorithms. For example, if we see the mean of a case as the sum of 3 3-cases-columns, we can consider that the value at $(i+1, j)$ is the value at (i, j) minus the leftmost column and plus the rightmost column. With that in mind, we can keep a buffer of 3 columns, and reduce or add them to next cases. This will increase the usefulness of the cache.

1.4 Conclusion

We have seen multiple algorithm and possible improvements :

- Switching
- Slicing
- Memory rearrangement

Each of them adds an improvement to the computation, the most effective being the memory rearrangement. To simplify the reasoning, we could have done it for the `quarter` or `half` methods, but the results wouldn't have been better (at most similar).