Report MPI Assignment

Author:
Gabriele Dellepere - S4944557

Head of analysis

Algorithm: the code provided has linear (N) complexity over the number of INTERVALS.
Tools: I used the *icx* and the *mpiicx* compilers and Intel Advisor GUI to perform the most of the analysis.
Machine: the machine on which we run the code has 20 processors:

- 12 core
    - 8 performance cores with hyperthreading up to 2
    - 4 efficiency cores

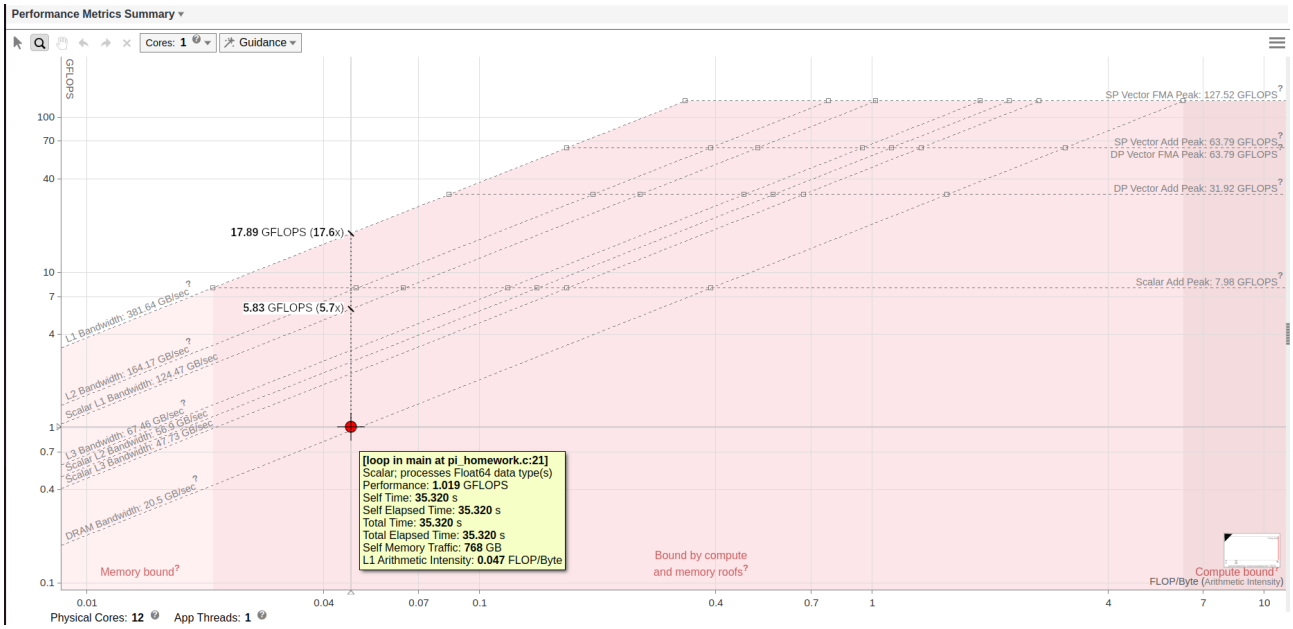
The program tries to approximate the value of pi using a tailor expansion on the equation of a circumference. The precision of the approximation seems to be determined by the value of the parameter INTERVALS, which determines the number of terms to sum up.

Hotspot identification

First thing I did was compiling the program and launching it, then feed the executable to intel advisor to obtain a detailed analysis.

```
- compiling line: icx -g -DINTERVALS=6000000000 pi_homework.c -o plain_icx_6000000000
- data size: INTERVALS=6000000000
- computed pi   : 3.14159265358923844857 4861
- true pi       : 3.14159265358979311599 7963
- time taken: 35.32 seconds
```

Advisor identified the loop that can be found at line 22 as the Hotspot, since it took 35.319 seconds out of the 35.320 total (everything else literally took 1 ms).



As we can see from the roofline model, the L1 arithmetic intensity of the program is quite low, hence the throughtput in terms of FLOPs is far from the Scalar add theoretical peak performance (and the computation is mainly held back by memory badwidth). If we look at the code, this result can be explained by manually computing the arithmetic complexity of the loop block.

```
for (i = 1; i <= intervals; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);
    sum = sum + f;
}
```

At each iteration, we have 6 floating point operations, but the amount of bytes accessed (either as read or write) is: 2*8 for x (written and read once per iteration) + 8 for dx (written once per iteration) 2*8 for f (written and read once per iteration), 2*8 for sum (written and read once per iteration) + 8 for intervals (read once per iteration) + AT LEAST 2*8 for i (could be much more depending on how smart/dumb the compiler is) → in the best case scenario, the arithmetic intensity would be about 0.075 FLOPs/byte. We can see from the roofline that it is even lower (0.047 FLOPs/byte), so the program is likely doing some inefficient loads/stores.

I proceeded with enabling compiler optimization and increasing the ITERATION amount in order to get once again an elapsed time greater than 30 seconds:

- **compiling line**: `icx -g -O3 -DINTERVALS=6000000000 pi_homework.c -o optimized_icx_60000000000`
- **data size**: `INTERVALS=6000000000`
- **computed pi**   : `3.14159265358980377413900`
- **true pi**   : `3.14159265358979311599796`
- **time taken**: `4.66 seconds`

```
- compiling line: icx -g -O3 -DINTERVALS=55000000000 pi_homework.c -o
optimized_icx_55000000000
- data size: INTERVALS=55000000000
- computed pi    : 3.14159265358966655057 3156
- true pi        : 3.14159265358979311599 7963
- time taken: 42.74 seconds
```

Vectorization issues

At this point, it was time to enable the intel compiler to exploit vectorization:

```
- compiling line: icx -g -O3 -xHost -qopt-report=3 -
DINTERVALS=55000000000 pi_homework.c -o vect_icx_55000000000
```

The report showed that there were no issues and the compiler managed to vectorize the loop at line 22, estimating a scalar cost of 23.00, a vector cost of 12.25, a potential speedup of 1.875 and a vectorization overhead of 0.5625. I then proceeded to launch the newly produced executable.

```
- data size: INTERVALS=55000000000
- computed pi    : 3.14159265358969186365 8118
- true pi        : 3.14159265358979311599 7963
- time taken: 31.99 seconds
```

We may observe that:

32 * (1.875 – 0.5625) ~= 42 → the report estimation was on point

Best sequential time

This time defining the best sequential time was fairly easy; I simply picked the execution time of *`vect_icx_55000000000`*

```
- time taken: 31.99 seconds
```

MPI Parallelization

To parallelize the computation, I only needed to delegate the partial estimation of “sum” to my processes. Luckily, the sum was just a single double, meaning that by leveraging the MPI_Reduce function I didn’t even need to allocate a source and destination buffer: a second double to hold the result was enough. The only part that required caution was splitting the

amount of work as equally as possible, while keeping in mind that it was not guaranteed that ITERATIONS would be divisible by the number of processes.

At the end, the code looked like this:

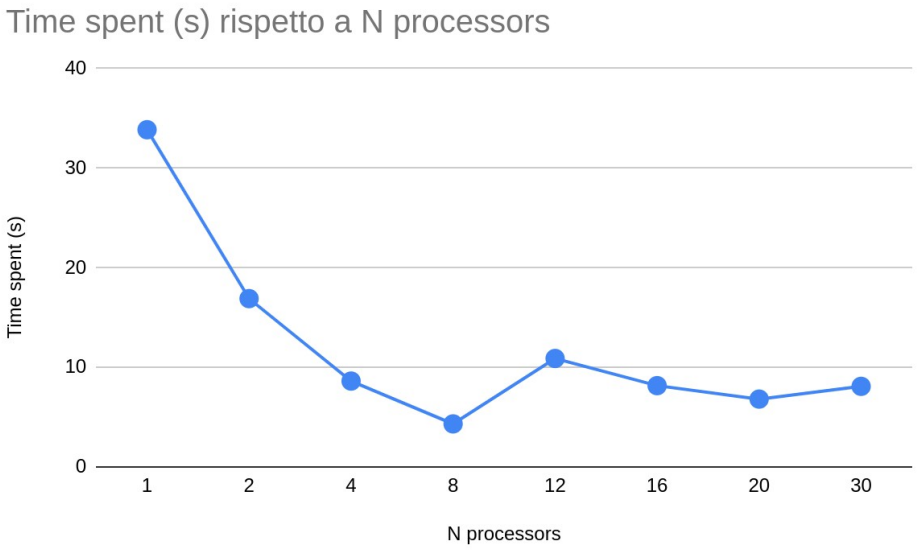
```
sum = 0.0;
dx = 1.0 / (double) intervals;

int rank;
int size;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

long work = intervals / size;
int remainder = intervals % size;

if (rank < remainder) { // then increase this process personal work by 1 and set its starting
    work++;
    i = rank * work;
} else {
    i = rank * work + remainder; // == (remainder * (work + 1)) + (rank - remainder) * work;
}

long limit = i + work;

for (; i < limit; i++) {
    // changed this to + 0.5 because it's simpler having i start from 0
    x = dx * ((double) (i + 0.5));
    f = 4.0 / (1.0 + x*x);
    sum = sum + f;
}

MPI_Reduce(&sum, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    pi = dx*result;

    time2 = (clock() - time1) / (double) CLOCKS_PER_SEC;

    printf("Computed PI %.24f\n", pi);
    printf("The true PI %.24f\n\n", PI25DT);
    printf("Elapsed time (s) = %.2lf\n", time2);
}

MPI_Finalize();
```

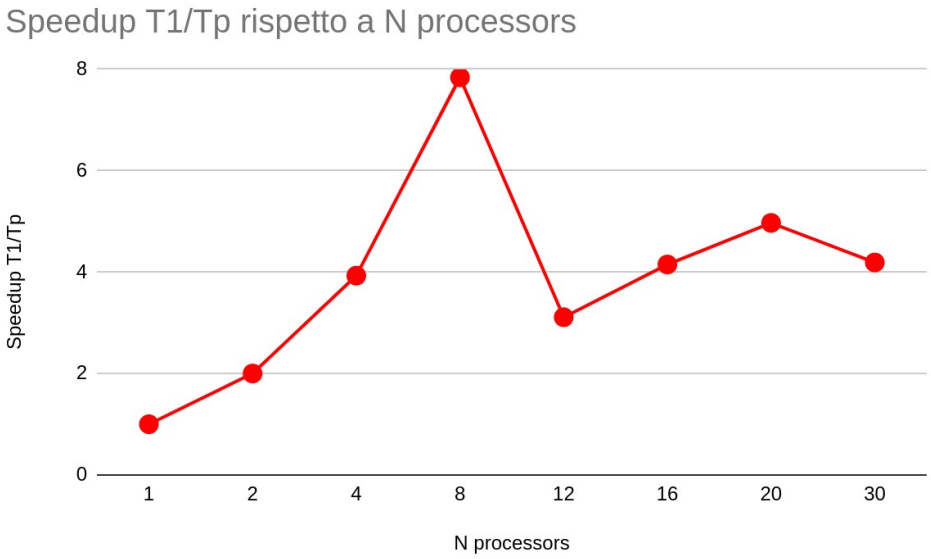
The program was compiled once and then launched multiple times on a different number of processes to gather the results.

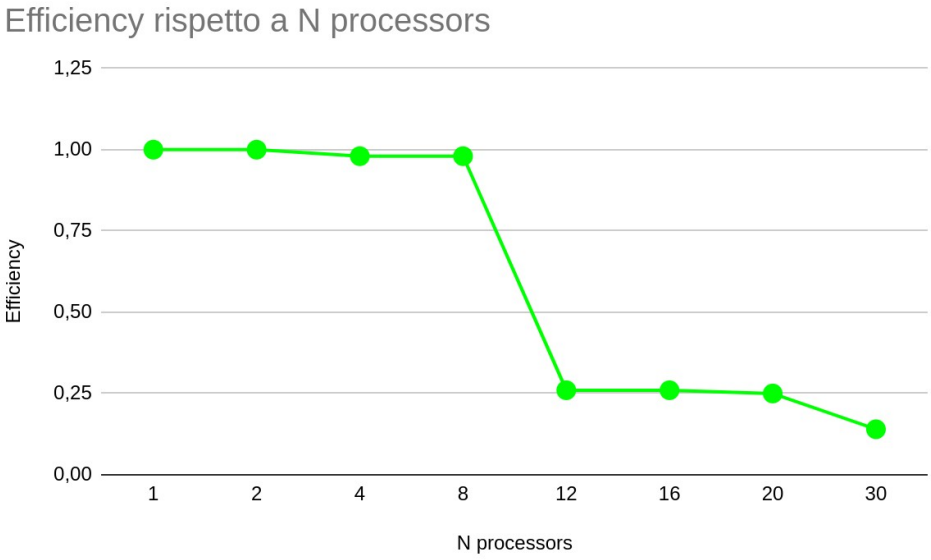
- **compiling line**: `mpiicx -g -O3 -xHost -DINTERVALS=55000000000 mpi_homework.c -o mpiicx_55000000000`
- **executing line**: `mpiexec -np <num_processes> mpiicx_55000000000`

The reason why there is no host file in the executing line (nor a limit to the amount of processes per host) is because unfortunately I had to launch all the processes on the same machine.
It follows a table of results, in terms of elapsed time, speedup and efficiency, for different values of <*num_processes*>. For each of this executions, I ensured that the error on the computed pi was never greater than 10^-12

Data size	`INTERVALS = 55000000000`		
N processors	Time spent (s)	Speedup T1/Tp	Efficiency
1	33.86	1.00	1.00
2	16.90	2.00	1.00
4	8.62	3.93	0.98
8	4.32	7.84	0.98
12	10.89	3.11	0.26
16	8.16	4.15	0.26
20	6.81	4.97	0.25
30	8.09	4.19	0.14







Conclusions

Given the results, there's no debate that, among the tried configurations, 8 processes is the optimal amount to use. What's interesting is understanding why is that so. The machine used for testing has 8 performance cores, which do support hyperthreading, but, differently from OpenMP, MPI works with processes, not threads, meaning that any amount of processes greater than 8 will force them to compete for the same hardware resources. There are the 4 efficiency cores available of course, but these ones are usually much slower than performance ones, and since the parallelized task requires the "root" process to wait for messages from all other processes, the final throughtput is determined by the slowest process in the communicator. Hence, any amount of processes greater than 8 on the same machine, for that particular machine, would cause a significant drop in performance.