Assignment II

Foundations of High Performance Computing

December 18, 2019

Eros Fabrici

Exercise 0

Measure the time-to-solution of the two codes in a strong-scaling test

The test were made with $N = 10^{10}$ and $N_c \in \{2, 3, 4, ..., 20\}$ on *Ulysses*. In Figure 1 it is possible to observe that `04_touch_by_all.c` scales better than the `01_array_sum.c`. This is due to the fact that in `01_array_sum.c` the array is allocated by the master thread, therefore the data is allocated in the area of the physical memory which is the closest to the core in which the before mentioned thread is executed. This leads to an non-negligible overhead, as, in the next for loop, the threads will have to remotely access the data located close to the master thread.

At the contrary, `04_touch_by_all.c` does not suffer from this as the data are allocated by using `pragma omp parallel for`, namely with a *touch-by-all* policy. As a consequence, the cache of each thread is warmed-up with the data that it will use after.

```python
import numpy as np
import matplotlib.pyplot as plt
# N = 10**10

#01_array_sum.c STRONG SCALING

T_1 = 3.93984

def speedup(t_1,T_p):
    return t_1/T_p

# (num_threads, speedup)

arraySumStrongScaling = [
    (2, speedup(T_1,1.99771)), (3, speedup(T_1,1.37858)),
    (4, speedup(T_1,1.03177)), (5, speedup(T_1,0.849771)),
    (6, speedup(T_1,0.716874)), (7, speedup(T_1,0.648061)),
    (8, speedup(T_1,0.600554)), (9, speedup(T_1,0.638904)),
    (10, speedup(T_1,0.636551)), (11, speedup(T_1,0.617983)),
    (12, speedup(T_1,0.596537)), (13, speedup(T_1,0.582717)),
    (14, speedup(T_1,0.607069)), (15, speedup(T_1,0.590117)),
    (16, speedup(T_1,0.597862)), (17, speedup(T_1,0.613692)),
    (18, speedup(T_1,0.59637)), (19, speedup(T_1,0.589231)),
    (20, speedup(T_1,0.605813))]

#04_touch_by_all.c STRONG SCALING

T_1 = 3.93805

touchByAllStrongScaling = [
    (2, speedup(T_1,1.99619)), (3, speedup(T_1,1.36036)),
    (4, speedup(T_1,1.02237)), (5, speedup(T_1,0.851516)),
    (6, speedup(T_1,0.704925)), (7, speedup(T_1,0.619455)),
    (8, speedup(T_1,0.544866)), (9, speedup(T_1,0.501635)),
    (10, speedup(T_1,0.452756)), (11, speedup(T_1,0.418485)),
    (12, speedup(T_1,0.384354)), (13, speedup(T_1,0.355025)),
    (14, speedup(T_1,0.332248)), (15, speedup(T_1,0.309369)),
    (16, speedup(T_1,0.293364)), (17, speedup(T_1,0.280248)),
```

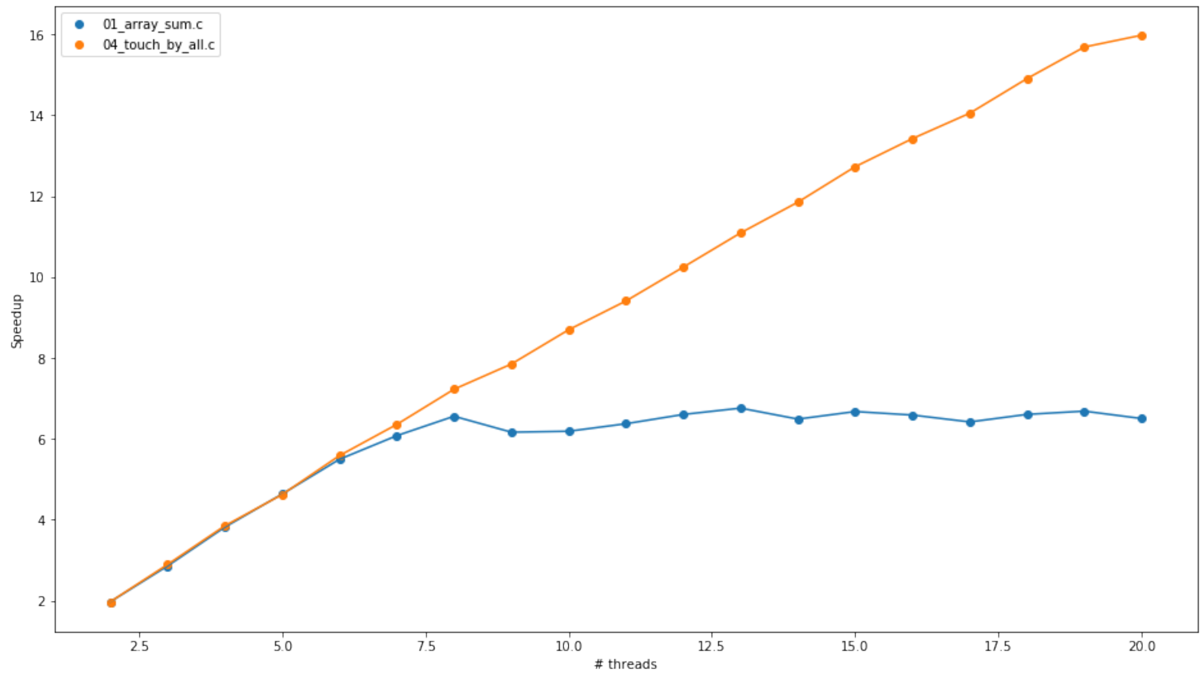

Figure 1: Strong Scaling on 01_array_sum.c and 04_touch_by_all.c

```
    (18, speedup(T_1,0.264153)), (19, speedup(T_1,0.251024)),
    (20, speedup(T_1,0.24643)) ]

#PLOTTING
plt.scatter(*zip(*arraySumStrongScaling), label= "01_array_sum.c")
plt.plot(*zip(*arraySumStrongScaling))

plt.scatter(*zip(*touchByAllStrongScaling), label= "04_touch_by_all.c")
plt.legend()
plt.xlabel("# threads")
plt.ylabel("Speedup")
plt.plot(*zip(*touchByAllStrongScaling))
plt.rcParams["figure.figsize"] = [16,9]
```

Measure the parallel overhead of both codes, from 2 to N_c cores on a node

As already said in the previous section, the bad scaling for the `01_array_sum.c` is due to a *communication overhead*, as the threads are accessing the memory close to a single core. In order to analyze better this phenomena I will use $E(p) = (1/Sp(p) - 1/p)/(1 - 1/p)$ (which we saw during the lectures). Table 1 shows clearly that the serial fraction, i.e. $E(p)$ start increasing when $p = 8$, and this is obviously due to the communication overhead. At the contrary, table 2 shows that $E(p)$ is constant.

p	2	4	8	10	14	18	20
Sp(p)	1.9722	3.8185	6.5603	6.1894	6.4899	6.6064	6.5034
E(p)	0.0141	0.0158	0.0313	0.0684	0.089	0.1014	0.1092

Table 1: Parallel overhead 01_array_sum.c

```
Samples: 20K of event 'branch-misses', Event count (approx.): 592472
+  49,06%  01_array_sum_pa  [kernel.kallsyms]
+  48,19%  01_array_sum_pa  01_array_sum_parallel.x
+   2,08%  01_array_sum_pa  libgomp.so.1.0.0
+   0,48%  01_array_sum_pa  ld-2.12.so
+   0,10%  01_array_sum_pa  libc-2.12.so
+   0,08%  01_array_sum_pa  [vdso]
Samples: 51K of event 'cache-misses', Event count (approx.): 210922708
+  84,46%  01_array_sum_pa  [kernel.kallsyms]
+  15,54%  01_array_sum_pa  01_array_sum_parallel.x
+   0,00%  01_array_sum_pa  libgomp.so.1.0.0
+   0,00%  01_array_sum_pa  ld-2.12.so
+   0,00%  01_array_sum_pa  libpthread-2.12.so
+   0,00%  01_array_sum_pa  libc-2.12.so
+   0,00%  01_array_sum_pa  [vdso]
Samples: 58K of event 'cycles', Event count (approx.): 46371246727
+  81,29%  01_array_sum_pa  01_array_sum_parallel.x
+  17,78%  01_array_sum_pa  [kernel.kallsyms]
+   0,93%  01_array_sum_pa  libgomp.so.1.0.0
+   0,00%  01_array_sum_pa  ld-2.12.so
Samples: 53K of event 'instructions', Event count (approx.): 38618680952
+  98,68%  01_array_sum_pa  01_array_sum_parallel.x
+   0,84%  01_array_sum_pa  [kernel.kallsyms]
+   0,48%  01_array_sum_pa  libgomp.so.1.0.0
+   0,00%  01_array_sum_pa  ld-2.12.so
+   0,00%  01_array_sum_pa  libc-2.12.so
+   0,00%  01_array_sum_pa  libpthread-2.12.so
```

Figure 2: array_sum - perf

p	2	4	8	10	14	18	20
Sp(p)	1.9728	3.8519	7.2276	8.698	11.8527	14.9082	15.9804
E(p)	0.0138	0.0128	0.0153	0.0166	0.0139	0.0122	0.0132

Table 2: Parallel overhead 04_touch_by_all.c

Provide any relevant metrics that explain any observed difference

In Figure 2 and 3 it is possible to observe that the `touch_by_all` program has way less cache misses than the other, as expected. In addition, `touch_by_all` has less cycles events.

Exercise 2

Tests were made with $data.length = 10^9$ and $search.length = 10^9$ and $N_c \in 2, 3, 4, ..., 20$ on *Ulysses*. I tested both the code with pre-fetching and without. Figure 4 shows that the one with pre-fetching scales better, while the other scales good up to ten threads. This is due to the fact that in the pre-fetching version at each iteration of the while loop, we pre-load the data to be compared in the cache.

Following the code for the plotting and with the times recorded.

```
#EXERCISE 2 - parallel Binary search
```

```
Samples: 20K of event 'branch-misses', Event count (approx.): 691581
+  62,86%  04_touch_by_all  [kernel.kallsyms]
+  35,28%  04_touch_by_all  04_touch_by_all.x
+   0,71%  04_touch_by_all  ld-2.12.so
+   0,63%  04_touch_by_all  libgomp.so.1.0.0
+   0,50%  04_touch_by_all  libc-2.12.so
+   0,03%  04_touch_by_all  libpthread-2.12.so
Samples: 41K of event 'cache-misses', Event count (approx.): 196003987
+  91,05%  04_touch_by_all  [kernel.kallsyms]
+   8,95%  04_touch_by_all  04_touch_by_all.x
+   0,00%  04_touch_by_all  libgomp.so.1.0.0
+   0,00%  04_touch_by_all  libc-2.12.so
+   0,00%  04_touch_by_all  ld-2.12.so
+   0,00%  04_touch_by_all  libpthread-2.12.so
Samples: 53K of event 'cycles', Event count (approx.): 40066278572
+  65,14%  04_touch_by_all  04_touch_by_all.x
+  33,99%  04_touch_by_all  [kernel.kallsyms]
+   0,87%  04_touch_by_all  libgomp.so.1.0.0
+   0,00%  04_touch_by_all  ld-2.12.so
Samples: 41K of event 'instructions', Event count (approx.): 38566644970
+  98,77%  04_touch_by_all  04_touch_by_all.x
+   0,86%  04_touch_by_all  [kernel.kallsyms]
+   0,37%  04_touch_by_all  libgomp.so.1.0.0
+   0,00%  04_touch_by_all  libpthread-2.12.so
+   0,00%  04_touch_by_all  ld-2.12.so
+   0,00%  04_touch_by_all  libc-2.12.so
```

Figure 3: touch_by_all - perf

```
#input: data = 10^9, search = 10^9

#serial binary search elapsed time
T_1 = 602.986

#(num_threads, elapsed_time)
bsStrongScal =
    [(2, speedup(T_1, 403.806)), (3, speedup(T_1, 290.276)),
     (4, speedup(T_1, 228.438)), (5, speedup(T_1, 190.084)),
     (6, speedup(T_1, 176.584)), (7, speedup(T_1, 158.189)),
     (8, speedup(T_1, 151.91)), (9, speedup(T_1, 141.307)),
     (10, speedup(T_1, 125.254)), (11, speedup(T_1, 140.749)),
     (12, speedup(T_1, 127.663)), (13, speedup(T_1, 132.639)),
     (14, speedup(T_1, 119.102)), (15, speedup(T_1, 127.597)),
     (16, speedup(T_1, 120.699)), (17, speedup(T_1, 124.852)),
     (18, speedup(T_1, 119.794)), (19, speedup(T_1, 123.941)),
     (20, speedup(T_1, 120.044))]

bs_prefetchStrongScal =
```

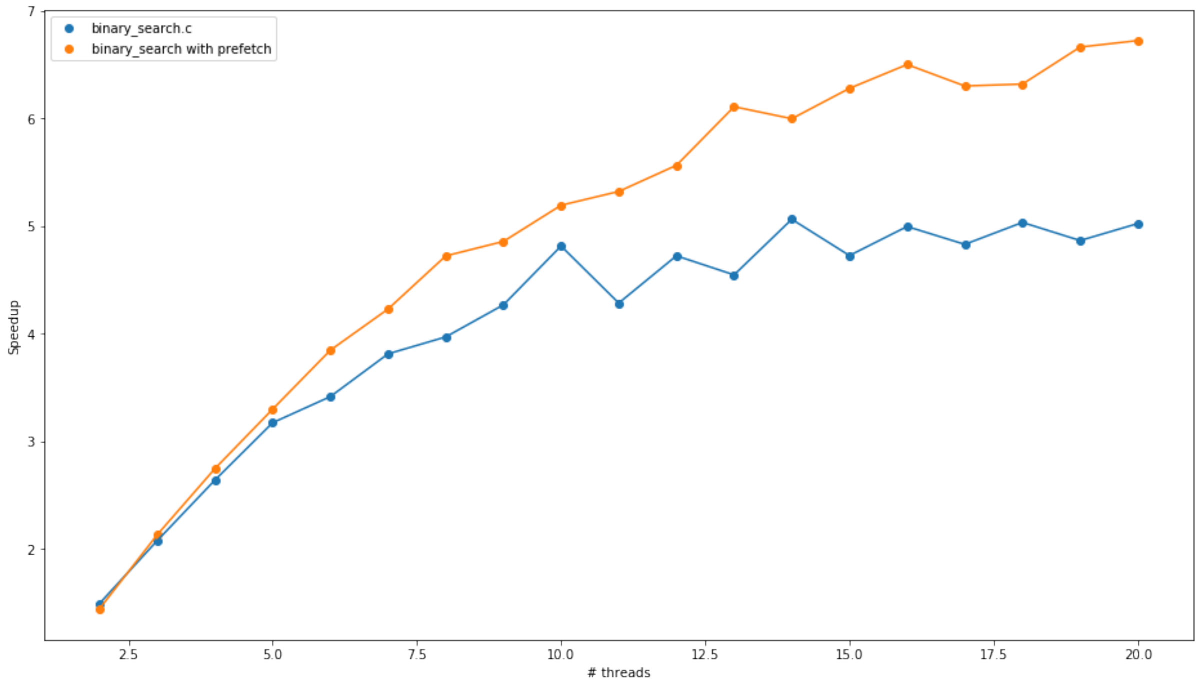

Figure 4: Strong Scaling binary_search.c

```
    [(2, speedup(T_1, 394.796)), (3, speedup(T_1, 267.578)),
     (4, speedup(T_1, 207.721)), (5, speedup(T_1, 172.965)),
     (6, speedup(T_1, 148.338)), (7, speedup(T_1, 134.922)),
     (8, speedup(T_1, 120.815)), (9, speedup(T_1, 117.472)),
     (10, speedup(T_1, 109.865)), (11, speedup(T_1, 107.212)),
     (12, speedup(T_1, 102.536)), (13, speedup(T_1, 93.3906)),
     (14, speedup(T_1, 95.1246)), (15, speedup(T_1, 90.853)),
     (16, speedup(T_1, 87.7581)), (17, speedup(T_1, 90.5363)),
     (18, speedup(T_1, 90.2816)), (19, speedup(T_1, 85.6217)),
     (20, speedup(T_1, 84.8532))]

#PLOTTING
plt.scatter(*zip(*bsStrongScal), label= "binary_search.c")
plt.plot(*zip(*bsStrongScal))
plt.scatter(*zip(*bs_prefetchStrongScal), label = "binary_search with prefetch")
plt.plot(*zip(*bs_prefetchStrongScal))
plt.legend()
plt.xlabel("# threads")
plt.ylabel("Speedup")
plt.rcParams["figure.figsize"] = [16,9]
```

Parallel Overhead

For `binary_search.c` $e(p)$ is stable with the growth of p, but it is definitely higher than the `touch_by_all.c` case. This is probably the cause that led to a stabilization of the speedup when p is bigger than 10.

Table 4 lists the results of the binary search with pre-fetching. It is clear that $e(p)$ stabilizes when $p \geq 10$, but it is lower than the one showed in Table 3.

```
Samples: 16M of event 'branch-misses', Event count (approx.): 16254235150
+  95,77%  binary_search_p  binary_search_parallel.x
+   2,93%  binary_search_p  [kernel.kallsyms]
+   1,29%  binary_search_p  libc-2.12.so
+   0,00%  binary_search_p  ld-2.12.so
+   0,00%  binary_search_p  libgomp.so.1.0.0
+   0,00%  binary_search_p  libpthread-2.12.so

Samples: 15M of event 'cache-misses', Event count (approx.): 43046038600
+  94,82%  binary_search_p  binary_search_parallel_prefetch.x
+   3,99%  binary_search_p  [kernel.kallsyms]
+   1,19%  binary_search_p  libc-2.12.so
+   0,00%  binary_search_p  libgomp.so.1.0.0
+   0,00%  binary_search_p  ld-2.12.so
+   0,00%  binary_search_p  libpthread-2.12.so

Samples: 17M of event 'instructions', Event count (approx.): 2390056526052
+  54,33%  binary_search_p  binary_search_parallel.x
+  42,51%  binary_search_p  [kernel.kallsyms]
+   3,14%  binary_search_p  libc-2.12.so
+   0,01%  binary_search_p  libgomp.so.1.0.0
+   0,00%  binary_search_p  ld-2.12.so

Samples: 17M of event 'cycles', Event count (approx.): 12924224996262
+  59,27%  binary_search_p  [kernel.kallsyms]
+  39,14%  binary_search_p  binary_search_parallel.x
+   1,59%  binary_search_p  libc-2.12.so
-   0,00%  binary_search_p  libgomp.so.1.0.0
     gomp_team_barrier_wait_end
   + gomp_barrier_wait_end
```

Figure 5: Binary search with prefetching - perf

p	2	4	8	10	14	18	20
Sp(p)	1.4933	2.6396	3.9694	4.8141	5.0628	5.0335	5.023
E(p)	0.3394	0.1718	0.1451	0.1197	0.1358	0.1515	0.1569

Table 3: Parallel overhead binary_search.c

p	2	4	8	10	14	18	20
Sp(p)	1.4451	2.7466	4.7223	5.193	5.9977	6.3194	6.7237
E(p)	0.384	0.1521	0.0992	0.1029	0.1026	0.1087	0.1039

Table 4: Parallel overhead binary_search.c with prefetching

However, both codes do not suffer of parallel an heavy overhead as in the `array_sum.c` program of the previous exercise. This is thanks to the fact we are using a “touch-by-all” policy, both for the `data` and `search` arrays.

Perf

Note: both Figure 5 and 6 have the same executable name, but they are not the same: 5 refers to the executable with the pre-fetching ON, while the other without pre-fetching.

```
Samples: 17M of event 'branch-misses', Event count (approx.): 16294556848
+  95,90%  binary_search_p  binary_search_parallel.x
+   2,78%  binary_search_p  [kernel.kallsyms]
+   1,32%  binary_search_p  libc-2.12.so
+   0,00%  binary_search_p  ld-2.12.so
+   0,00%  binary_search_p  libgomp.so.1.0.0
```

```
Samples: 17M of event 'cache-misses', Event count (approx.): 40429391406
+  95,27%  binary_search_p  binary_search_parallel.x
+   3,68%  binary_search_p  [kernel.kallsyms]
+   1,06%  binary_search_p  libc-2.12.so
+   0,00%  binary_search_p  ld-2.12.so
+   0,00%  binary_search_p  libgomp.so.1.0.0
+   0,00%  binary_search_p  libpthread-2.12.so
```

```
Samples: 18M of event 'instructions', Event count (approx.): 1801159962575
+  59,04%  binary_search_p  [kernel.kallsyms]
+  36,78%  binary_search_p  binary_search_parallel.x
+   4,16%  binary_search_p  libc-2.12.so
+   0,01%  binary_search_p  libgomp.so.1.0.0
```

```
Samples: 18M of event 'cycles', Event count (approx.): 13879806637478
+  56,47%  binary_search_p  [kernel.kallsyms]
+  42,03%  binary_search_p  binary_search_parallel.x
+   1,50%  binary_search_p  libc-2.12.so
+   0,00%  binary_search_p  libgomp.so.1.0.0
+   0,00%  binary_search_p  ld-2.12.so
```

Figure 6: Binary Search (WITHOUT prefetching) - perf

Appendix

Everything was tested on a single node in *Ulysses*. Following the scripts used for running the programs and perf.

Exercise 0

Compiled with GCC 4.9.2

```
FHPC_2019-2020/Assignements/Assignment02/./01_array_sum.x 10000000000

module load gnu
for threads in 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ; do
    export OMP_NUM_THREADS=${threads}
    FHPC_2019-2020/Assignements/Assignment02/./01_array_sum_parallel.x 10000000000
done
```

```
module load gnu
for threads in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ; do
    export OMP_NUM_THREADS=${threads}
    FHPC_2019-2020/Assignements/Assignment02/./04_touch_by_all.x 10000000000
done
```

```
module load gnu
export OMP_NUM_THREADS=20
```

```
perf record --call-graph -e cycles,instructions,cache-misses,branch-misses
FHPC_2019-2020/Assignements/Assignment02/./01_array_sum_parallel.x 10000000000
```

```
module load gnu
export OMP_NUM_THREADS=20
perf record --call-graph -e cycles,instructions,cache-misses,branch-misses
FHPC_2019-2020/Assignements/Assignment02/./04_touch_by_all.x 10000000000
```

For generating the report I used the command `perf report --sort comm,dso`

Binary Search

Compile with GCC 4.9.2 at least, use flag `-lrt` for the serial one.

```
module load gnu
echo "SERIAL"
(FHPC_2019-2020/Assignements/Assignment02/./binary_search.x 1000000000 1000000000)
echo "PARALLEL"
for threads in 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ; do
    export OMP_NUM_THREADS=${threads}
    (FHPC_2019-2020/Assignements/Assignment02/./binary_search_parallel.x 1000000000
    1000000000)
done
```

```
module load gnu
echo "SERIAL"
(FHPC_2019-2020/Assignements/Assignment02/./binary_search_prefetch.x 1000000000 1000000000)
echo "PARALLEL"
for threads in 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ; do
    export OMP_NUM_THREADS=${threads}
    (FHPC_2019-2020/Assignements/Assignment02/./binary_search_parallel_prefetch.x 1000000000
    1000000000)
done
```

```
module load gnu
export OMP_NUM_THREADS=20
perf record --call-graph -e cycles,instructions,cache-misses,branch-misses
FHPC_2019-2020/Assignements/Assignment02/./binary_search_parallel.x 1000000000 1000000000
```

```
module load gnu
export OMP_NUM_THREADS=20
perf record --call-graph -e cycles,instructions,cache-misses,branch-misses
FHPC_2019-2020/Assignements/Assignment02/./binary_search_parallel_prefetch.x 1000000000 1000000000
```