Project 3: parallelized gravity models

Behnam Saeedi
(Saeedib@oregonstate.edu)
CS575: Parallel Programming

Spring 2018

Abstract

As covered in class there are two main fixes we can apply to combat false sharing. The two fixes have their advantages and disadvantages. In this project we will implement, analyze and compare the two different fixes to see how thee two fixes help us with the performance of our multi threaded operation.

Contents

1 System Specs

The computer used for running these experiments is A lenovo with i7 (Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz) with 8 GB of ram.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz
Stepping:              9
CPU MHz:               2793.686
CPU max MHz:           3100.0000
CPU min MHz:           800.0000
BogoMIPS:              4988.73
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              4096K
NUMA node0 CPU(s):     0-3
```

2 Approach

In order to create the timing needed for the performance of this project, a script was required to run our code 16 times per thread trial. This created an experiment that runs the program51 times. (48 times for FIX1 and 3 times for FIX2). We used the following g++ call to compile the program:

```
g++ -DPADDING=$padding -DFIX# -fopenmp OMPUtil.cpp main.cpp -o out -g -O0
```

This call does the following tasks:

- **-DPADDING=padding:** Defines and sets the number of padding needed. Please note this needs to be set in compile time since C Structures are static constructs.
- **-DFIX1/2:** Defines and sets which fix is going to be used.
- **-g flag:** This flag is responsible for creating the necessary overhead needed for GDB to help us debug our program in case of segmentation faults.
- **-O0 flag:** "-O" flag is the optimizer level specifier. We can pick how much optimization we need by calling flag -O and an integer. In this case we need to eliminate all optimization steps. This is done by passing "-O0" or level 0 optimization.

```
#ifdef FIX1
        struct s
        {
                float value;
                int pad[PADDING];
        } Array[4];
#endif
#ifdef FIX2
        struct s
        {
                float value;
        } Array[4];
#endif
//...
#ifdef FIX1
        Array[ i ].value = Array[ i ].value + 2.;
#endif
#ifdef FIX2
        float tmp = Array[ i ].value + 2.;
        Array[ i ].value = tmp;
#endif
```

We are using compiler directives to make sure our code is efficient during the run time. The flags passed to the compiler sets this pre-process directives allowing us to control which fix and how much padding to be applied. further more, we can see the fixes in the parallel for loop as well. Please note that depending on the directive the actual execution looks like

```
        struct s
        {
                float value;
                int pad[PADDING];
        } Array[4];
//...
        Array[ i ].value = Array[ i ].value + 2.;
```

or

```
        struct s
        {
                float value;
        } Array[4];
//...
        float tmp = Array[ i ].value + 2.;
        Array[ i ].value = tmp;
```

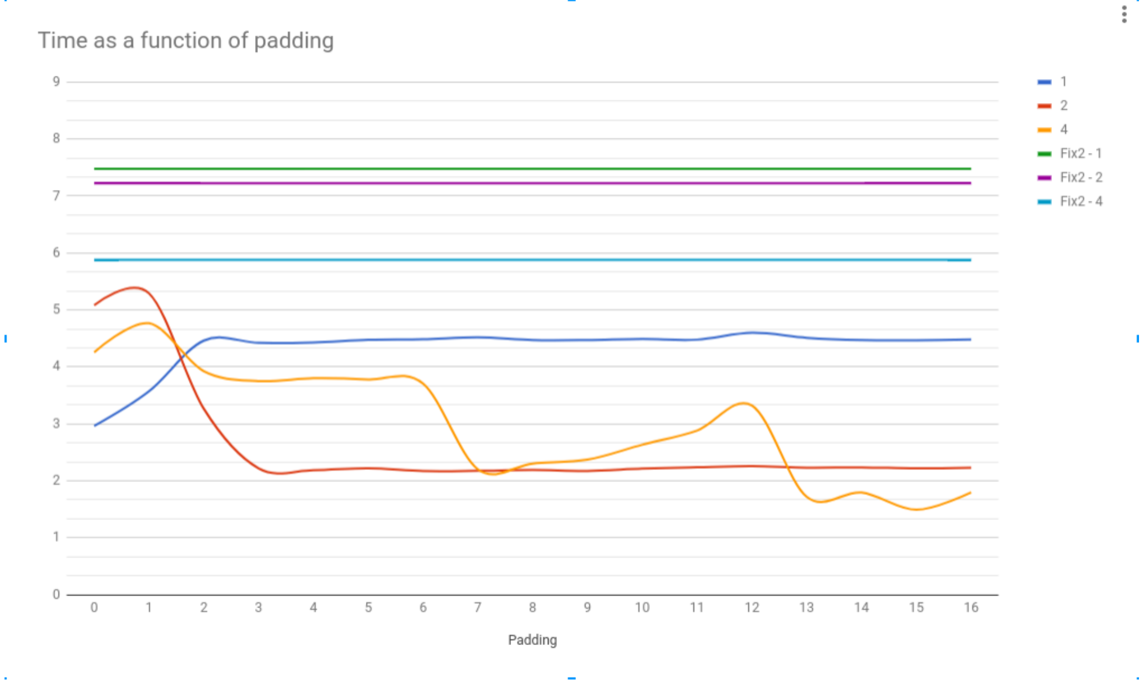
Finally a bash script was needed to automate the data collection process:

```
#!/bin/bash
for thread in `seq 0 2`; do
        echo "Using $[2**$thread] threads: "
        for padding in `seq 0 16`; do
                echo "Using $padding paddings: "
                g++ -DPADDING=$[$padding] -DFIX1 -fopenmp OMPUtil.cpp main.cpp -o out -g -O0
                ./out $[2**$thread] s >> tmp$[2**$thread]
                echo "">> tmp$[2**$thread]
```

```
            ./out $[2**$thread]
        done
        echo "Using FIX 2: "
        g++ -DPADDING=0 -DFIX2 -fopenmp OMPUtil.cpp main.cpp -o out -g -O0
        ./out $[2**$thread] s >> tmp$[2**$thread]
        ./out $[2**$thread]
done
```

3 Time Trials

This process ran for 48 times on different number of paddings and threads. The result of this experiment is present in Table 1. The general trend with Fix 1 is that as the number of padding increases, the program slows down and we see steady increase in the time. This is very clear in single thread performance (dark blue line). This trend continues on multi threaded executions as well all the way until the number of padding exceeds 3 for 2 threaded process and 7 for 4 threaded process. The reason to this is that after 3 paddings in 2 threaded and 7 paddings for 4 threaded, data falls on different cache lines. In two threaded process as soon as one falls on a different cache line we are done and things get much faster. With 4 treads however, optimal performance happens wen all f threads get their own cache line. After this improvement in performance, the time starts rising again. Things get slower because the process of allocating memory using malloc is linear time deterministic. (based on type of slab, it uses different functions to determine where the memory starts and ends and this has a large overhead as a function of size). The time keeps on rising until for the four threaded process it falls again at 14 paddings. here there is a full empty cache line space between each data. However, this improvement is not significant.



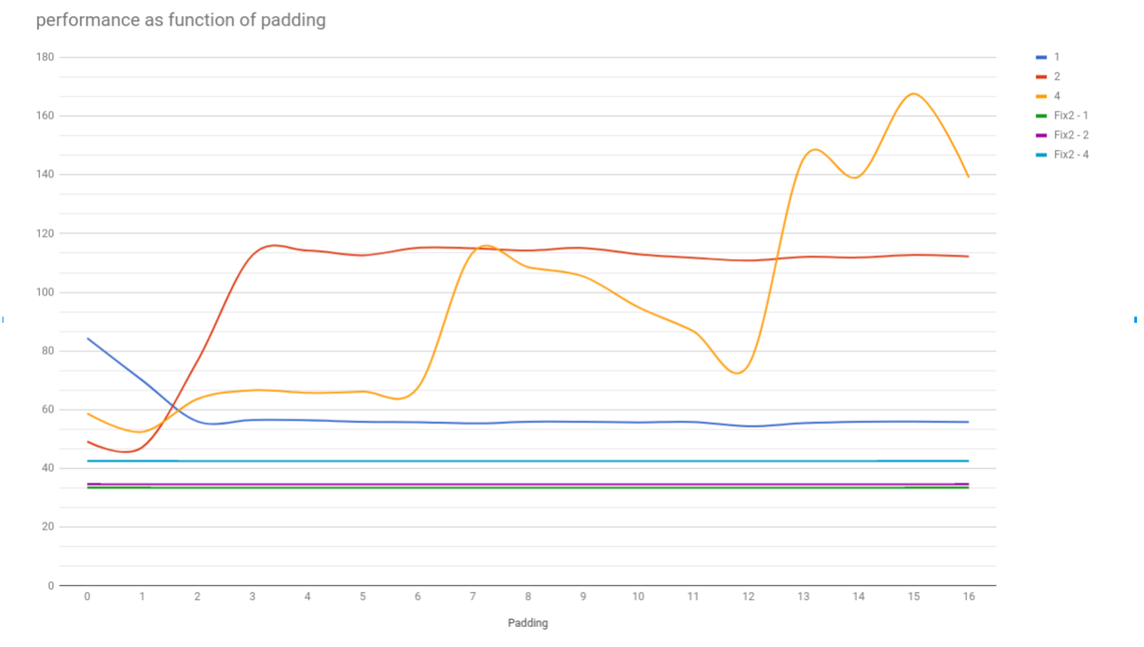
4 Performance

The performance is the number of operations divided be spent time. It is natural to see as time decreases the performance improves. here we can see this in action. On padding numbers 3 for two threaded and 7 and 14

Table 1
Time as a function of padding on number of threads

	1	2	4	Fix2-1	Fix2-2	Fix2-4
0	2.962301	5.083286	4.257364	7.477878	7.224013	5.879962
1	3.571564	5.291146	4.765132	7.477878	7.224013	5.879962
2	4.461777	3.263788	3.924718	7.477878	7.224013	5.879962
3	4.421227	2.218742	3.75131	7.477878	7.224013	5.879962
4	4.4296	2.188317	3.801355	7.477878	7.224013	5.879962
5	4.473255	2.220442	3.779739	7.477878	7.224013	5.879962
6	4.483138	2.171242	3.704057	7.477878	7.224013	5.879962
7	4.516657	2.174153	2.200506	7.477878	7.224013	5.879962
8	4.471394	2.188777	2.30236	7.477878	7.224013	5.879962
9	4.472073	2.17207	2.371107	7.477878	7.224013	5.879962
10	4.488969	2.212743	2.633417	7.477878	7.224013	5.879962
11	4.478028	2.237345	2.881428	7.477878	7.224013	5.879962
12	4.597553	2.256086	3.321337	7.477878	7.224013	5.879962
13	4.510469	2.230814	1.718223	7.477878	7.224013	5.879962
14	4.471714	2.235057	1.793673	7.477878	7.224013	5.879962
15	4.46768	2.217931	1.490896	7.477878	7.224013	5.879962
16	4.478518	2.22791	1.797659	7.477878	7.224013	5.879962

for four threaded process we see significant improvements in performance (in units of MegaOps per second).



5 Analysis of results

one thing to note is that the performance of the Fix 2 was consistently much lower than the performance of Fix 1 across all threads. The reason to this might be that the overhead for allocation a given amount of memory is

Table 2
Performance as a function of padding on number of threads

	1	2	4	Fix2-1	Fix2-2	Fix2-4
0	84.39385464	49.18078581	58.72178183	33.4319442	34.60680373	42.51728157
1	69.9973457	47.24874347	52.4644438	33.4319442	34.60680373	42.51728157
2	56.03148701	76.59811238	63.6988441	33.4319442	34.60680373	42.51728157
3	56.54538887	112.6764626	66.64338591	33.4319442	34.60680373	42.51728157
4	56.43850461	114.2430461	65.7660229	33.4319442	34.60680373	42.51728157
5	55.88771487	112.590196	66.14213309	33.4319442	34.60680373	42.51728157
6	55.76451138	115.141472	67.49356179	33.4319442	34.60680373	42.51728157
7	55.35067197	114.9873077	113.6102333	33.4319442	34.60680373	42.51728157
8	55.91097541	114.2190365	108.5842353	33.4319442	34.60680373	42.51728157
9	55.90248639	115.0975797	105.4359841	33.4319442	34.60680373	42.51728157
10	55.6920754	112.9819414	94.93369261	33.4319442	34.60680373	42.51728157
11	55.82814578	111.7395842	86.7625358	33.4319442	34.60680373	42.51728157
12	54.37675215	110.8113786	75.27089241	33.4319442	34.60680373	42.51728157
13	55.42660863	112.0667165	145.4991581	33.4319442	34.60680373	42.51728157
14	55.90697437	111.8539706	139.3788054	33.4319442	34.60680373	42.51728157
15	55.95745443	112.7176634	167.6843992	33.4319442	34.60680373	42.51728157
16	55.82203756	112.2127914	139.0697568	33.4319442	34.60680373	42.51728157

just too high for this problem to yield a reasonable result on a home computer. Using a different compiler or hardware might have yielded a significant improvement over mine and my home PC.

6 Conclusion

Based on my experiment on my personal computer I conclude that Fix 1 has a significantly better impact on the performance of a multi threaded process. The data collected suggests that a padding of correct size can allow threads to operate on their own cache lines and helps the program circumvent the false sharing fault.