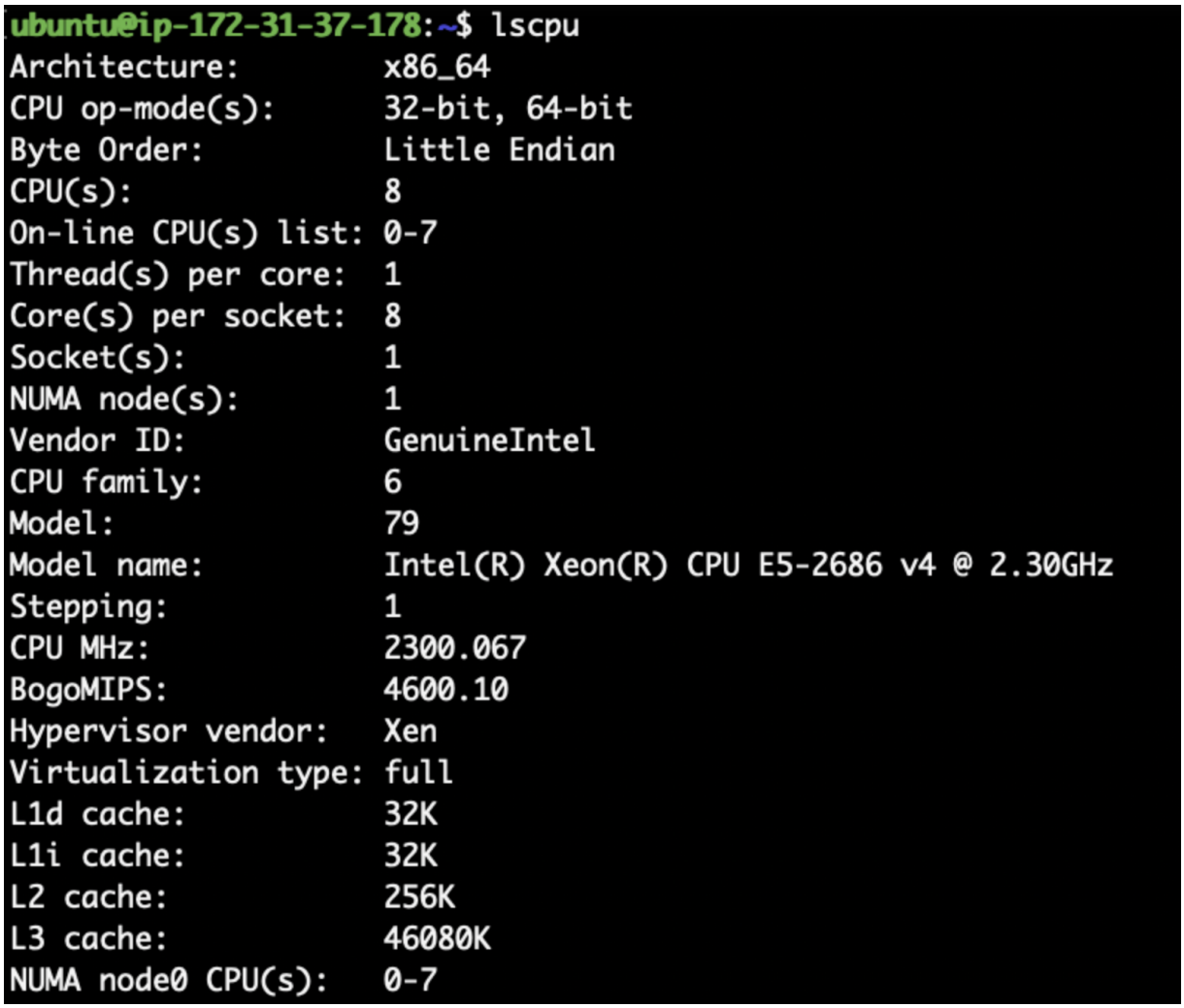
> By Jiahui Tang

Spec:

- OS: AWS VM Ubuntu 18.04 (x2.2large)
- Kernal: Linux ip-172-31-37-178 5.4.0-1039-aws #41~18.04.1-Ubuntu SMP Fri Feb 26 11:20:14 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
- Compiler: gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
- Flag: -Ofast
- Details:

```
ubuntu@ip-172-31-37-178:~$ uname -a
Linux ip-172-31-37-178 5.4.0-1039-aws #41~18.04.1-Ubuntu SMP Fri Feb 26 11:20:14 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
ubuntu@ip-172-31-37-178:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
stepping        : 1
microcode       : 0xb000038
cpu MHz         : 2299.914
cache size      : 46080 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflu
 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm cpuid_
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_mul
bogomips        : 4600.10
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```


ubuntu@ip-172-31-37-178:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
ubuntu@ip-172-31-37-178:~$


ubuntu@ip-172-31-37-178:~$ lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 1
Core(s) per socket: 8
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 79
Model name: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
Stepping: 1
CPU MHz: 2300.067
BogoMIPS: 4600.10
Hypervisor vendor: Xen
Virtualization type: full
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 46080K
NUMA node0 CPU(s): 0-7

4. Shared-Memory Parallel Processing (35 points)

4.1. Parallelization of Basic Code (20 points)

The goal of this exercise[2] is to familiarize you with OpenMP and make your first parallel code. Develop an OpenMP version of the jacobi code used in section 2.1 by completing `jacobi1d_omp.c`. Your task is to parallelize the `for` loops in the function jacobi using OpenMP. You should do this by copying over the corresponding code from `jacobi1d.c` and putting OpenMP `#pragma`s in appropriate places. You should start by putting a `#pragma parallel` inside the outer loop in order to start a parallel section, and then parallelize the inner loops with `#pragma for` directives.

You should do two experiments:

1. Run your parallel codes on a system with at least 4 cores with `ncells` equal to 10^k, for k = 5, 6, 7, and 8, and 100 steps. Plot the ratio of your parallel run times to the serial code run time as a function of k. Calculate the Speed-up and explain why it increases or decreases with the problem size. Use -Ofast to compile the code in all cases.
2. Run your codes with `ncells` of 10^8 and and 100 steps on 2-8 cores and produce a speedup plot. Explain why it increases or decreases with the number of cores.

You should check the correctness of your implementation by comparing the solution output files to the solution output files from the serial code.

Submission

- `P41.pdf`: Explain your results in terms of a simple performance model that takes into account the time spent on both computation (which is proportional to the number of points per processor) and communication/synchronization.
    - Hint: No need to present a mathematical model. Just describe using words. You can think about concepts such as granularity and synchronization overhead.
- `P41.c`: Complete source code

(4.1.1). Table for improvements in elapsed execution time

(*unit: seconds*)

With `ncells = 10^k` , `step=100` , `OMP_NUM_THREADS=8` , `-Ofast`

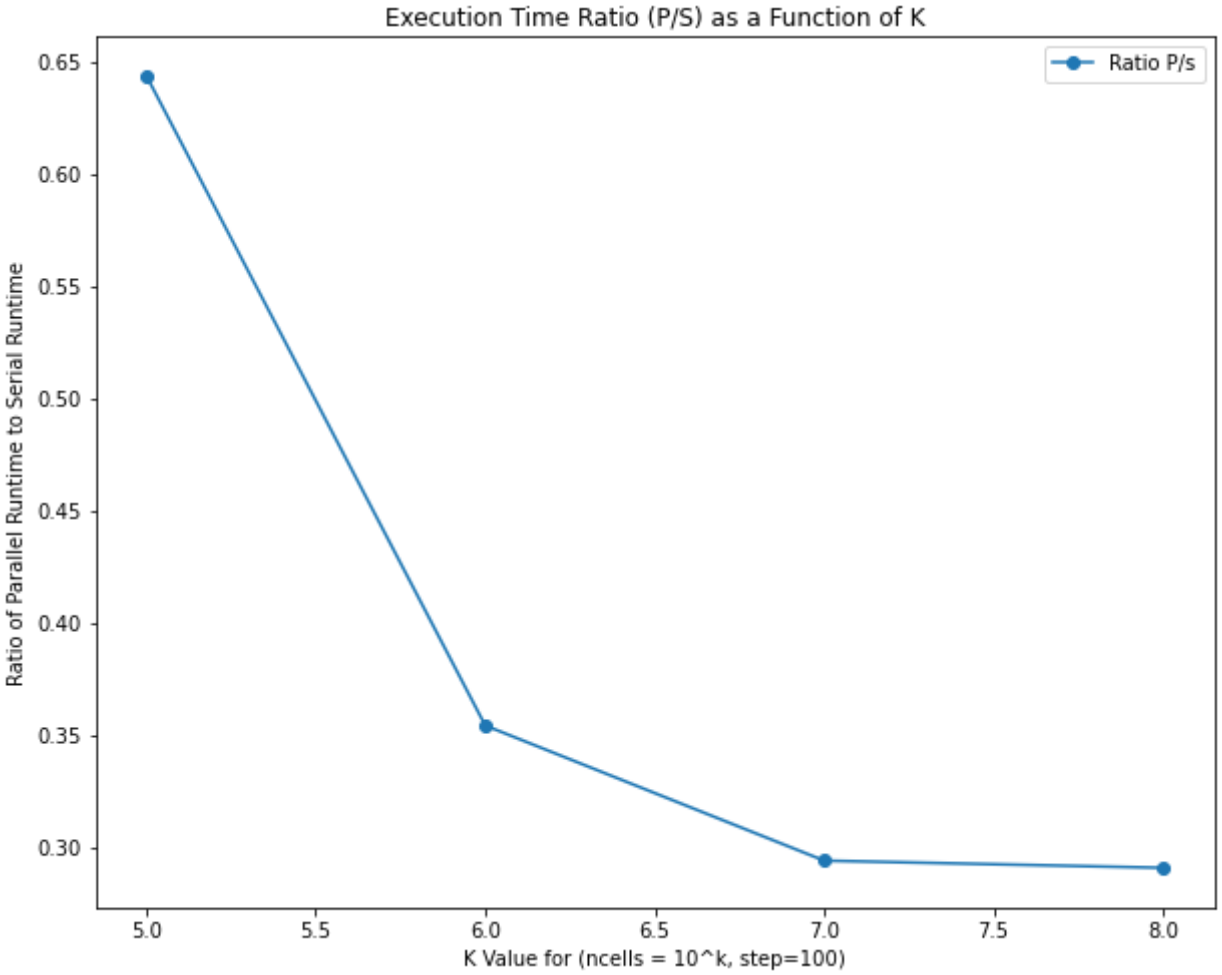
	k=5 Elpased Time	k=6 Elpased Time	k=7 Elpased Time	k=8 Elpased Time
Parallel Code Runtime	0.007	0.037	0.731	7.224
Sequential Code Runtime	0.010865	0.104475	2.48723	24.8462
Ratio P/S	0.6442	0.3542	0.2939	0.2907
Speedup S/P	1.5521	2.8236	3.4025	3.4394

In [3]:
```
import matplotlib.pyplot as plt

k = [5,6,7,8]
ratio = [0.6442,0.3542,0.2939,0.2907]
#speedup = [1.5521,2.8236,3.4025,3.4394]

plt.figure(figsize = (10,8))
plt.plot(k, ratio, '-ob', label = "Ratio P/s", color="tab:blue")
#plt.plot(k, speedup, '-ob', label = "Speedup S/P",color="tab:orange")

plt.xlabel('K Value for (ncells = 10^k, step=100)')
plt.ylabel('Ratio of Parallel Runtime to Serial Runtime')
plt.title('Execution Time Ratio (P/S) as a Function of K')
plt.legend()
plt.show()
```



Discussion:

- The speed up increases with problem size, from 1.5 to 3.4 when k increases from 5 to 8.
- In order to measure times we must use real time and not cpu time, which adds the time consumed by the process in all CPUs.
- Speedup increase with problem size because as we fixed number of thread, the overhead for synchronization and communication is roughly the same for all problem sizes. When K increases, the overheads will thus take a smaller and approaching neglible proportion of overall runtime. Total runtime adds up the parallel time and overhead time, thus it will result in a higher and higher speedup.
- It also suffer from a diminishing marginal increments because as problem size goes up, the overhead time become insignificant. The speedup thus gradually approaches real speedup.
- Also there's diminishing marginal incremental in speedup rate when problem size goes up, as `granularity= computation/communication ratio` , when fixing number of cores, changing problem size changes the computation portion of granularity. The granularity increased to a higher ratio, became `coarse-grained` , where relatively large amounts of computational work are done between communication/synchronization events. Thus, gradually, the `load imbalance overhead` may dominates overhead, and also lead to higher execution time. It slows down the acceleration of speedup, making the marginal increase became smaller as problem size goes up.

- Finally, the speedup follows Amdahl's Law, is defined by the fraction of code (c) that can be parallelized, and bounded by sequential code, even a small percentage of sequential code can greatly limit potential speedup. Because it has overhead, thus the speedup never reaches theory speedup of 4x but were approaching limit at around 3.5.

```
ubuntu@ip-172-31-37-178:~$ export OMP_NUM_THREADS=8
ubuntu@ip-172-31-37-178:~$ ./jacobi1d 100000 100
n: 100000
nsteps: 100
Elapsed time: 0.010093 s
ubuntu@ip-172-31-37-178:~$ ./jacobi1d 1000000 100
n: 1000000
nsteps: 100
Elapsed time: 0.104475 s
ubuntu@ip-172-31-37-178:~$ ./jacobi1d 10000000 100
n: 10000000
nsteps: 100
Elapsed time: 2.48723 s
ubuntu@ip-172-31-37-178:~$ ./jacobi1d 100000000 100
n: 100000000
nsteps: 100
Elapsed time: 24.8462 s
```

```
ubuntu@ip-172-31-37-178:~$ export OMP_NUM_THREADS=8
ubuntu@ip-172-31-37-178:~$ time ./omp_p41 100000 100 >output

real    0m0.007s
user    0m0.044s
sys     0m0.003s
ubuntu@ip-172-31-37-178:~$ time ./omp_p41 1000000 100 >output

real    0m0.037s
user    0m0.155s
sys     0m0.101s
ubuntu@ip-172-31-37-178:~$ time ./omp_p41 10000000 100 >output

real    0m0.731s
user    0m4.865s
sys     0m0.591s
ubuntu@ip-172-31-37-178:~$ time ./omp_p41 100000000 100 >output

real    0m7.224s
user    0m47.147s
sys     0m6.698s
ubuntu@ip-172-31-37-178:~$
```

(4.1.2). Table for improvements in elapsed execution time

(*unit: seconds*)

With `ncells = 10^8 , step=100 , -Ofast`

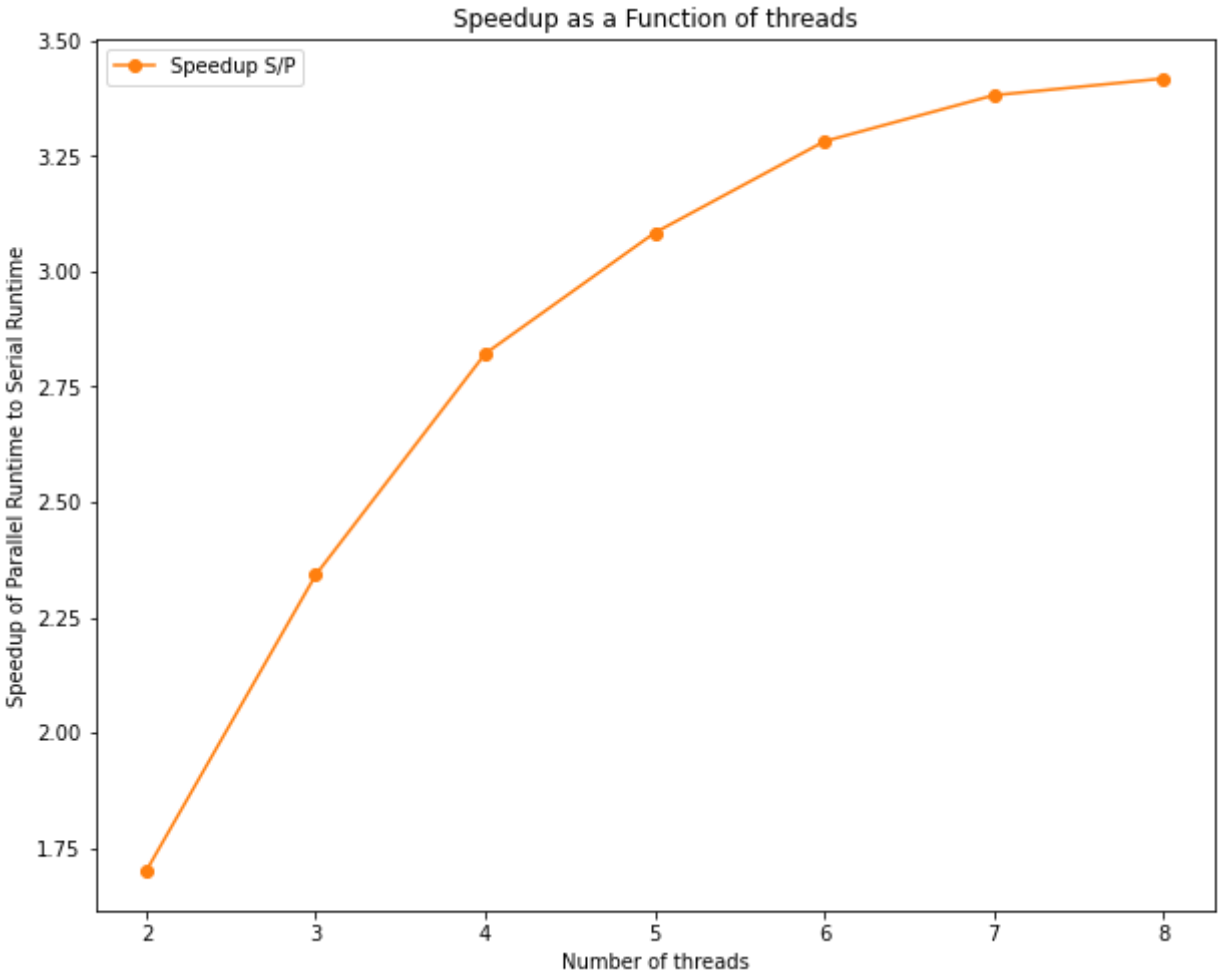
OMP_NUM_THREADS	2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores
Parallel Code Runtime	14.742	10.665	8.868	8.066	7.614	7.363	7.238
Sequential Code Runtime	25.0803	24.9811	25.0209	24.8697	24.9831	24.8953	24.7347
Ratio P/S	0.5878	0.4269	0.3544	0.3243	0.3048	0.2957	0.2926
Speedup S/P	1.7013	2.3423	2.8215	3.0833	3.2812	3.3811	3.4173

In [2]:
```
import matplotlib.pyplot as plt

k = [2,3,4,5,6,7,8]
ratio = [0.5878,0.4269,0.3544,0.3243,0.3048,0.2957,0.2926]
speedup = [ 1.7013,2.3423,2.8215,3.0833,3.2812,3.3811,3.4173]

plt.figure(figsize = (10,8))
#plt.plot(k, ratio, '-ob', label = "Ratio P/s", color="tab:blue")
plt.plot(k, speedup, '-ob', label = "Speedup S/P",color="tab:orange")

plt.xlabel('Number of threads')
plt.ylabel('Speedup of Parallel Runtime to Serial Runtime')
plt.title('Speedup as a Function of threads')
plt.legend()
plt.show()
```



Discussion/Performance Model:

The speedup increases with number of cores in shared memory parallel processing. It makes sense because as number of cores go up, computation speedup also increases with the number of cores/threads. More cores allow more concurrency when shared same memory, that could simultaneously execute statements in the parallel region.

But it has diminishing marginal return after `core=6` to `core=8`, it is probably due to time spent on communication/synchronization/load imbalance overhead.

Where `granularity= computation/communication ratio`, when the core goes up, keeping problem size fixed, the granularity decreased to a lower ratio, became `fine-grained`, communicate/sync dominates the overhead time. As the parallelized version introduces many communication overhead associated with the setup of the runtime environment and the creation of the thread. Also large number of cores requires more overhead time for synchronization, decreasing the speedup rate. Thus, it has a smaller marginal incremental of speedup rate as number of cores goes up.

Finally, the speedup follows Amdahl's Law, is defined by the fraction of code (c) that can be parallelized, and bounded by sequential code, even a small percentage of sequential code can greatly limit potential speedup. Because it has overhead, thus the speedup never reaches theory speedup.