CS263 Project Report

CS263 Project Report	Chang Lu	6384630
	Haowen Zhang	6425607

Python Interpreter Performance Comparison

1. Background

Python has so many Interpreters and each of them has different advantages and disadvantages compared to each other. We pick four Python Interpreters to compare their performance. We select CPython, IronPython, PyPy, and Jython to compare.

1.1 Interpreters

CPython is the reference implementation of Python, written in C. It compiles Python code to intermediate bytecode which is then interpreted by a virtual machine. CPython provides the highest level of compatibility with Python packages and C extension modules. PyPy is a Python interpreter implemented in a restricted statically-typed subset of the Python language called RPython. The interpreter features a just-in-time compiler. Jython is a Python implementation that compiles Python code to Java bytecode which is then executed by the JVM (Java Virtual Machine). Additionally, it is able to import and use any Java class like a Python module. IronPython is an implementation of Python for the .NET framework. It can use both Python and .NET framework libraries, and can also expose Python code to other languages in the .NET framework.

1.2 Benchmarks

We find our benchmarks on a website (https://pybenchmarks.org/). We select the programs that can be run on all the interpreters. The programs has different features so that we can compare the Interpreters in different tasks.

1.3 Implementation

We Implement the measurement on both Windows and Linux platform to see if there is any difference between platforms.

1.3.1 Psutil

We measure the program performance with a package called psutil. psutil (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. We measure the Elapsed time, CPU time, Memory usage, and CPU load of each program with different interpreters and collect the data.

The CPU time is returned by psutil.Process().cpu_times() when the child process is existing and the last data before the child process exists is the final result.

The time is taken before forking the child-process and after the child-process exits, using time.time() to calculate the Elapsed time.

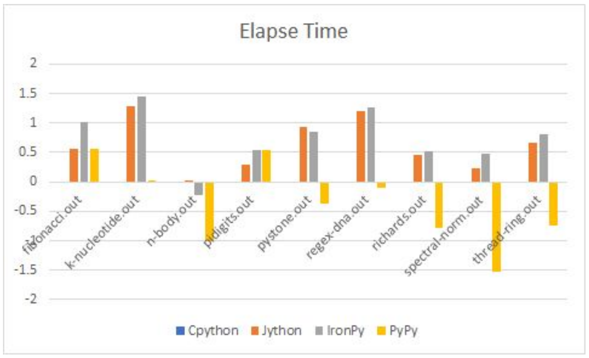
The Memory usage comes from psutil.Process().memory_info().rss continuously when the child process exists. the largest value returned is the memory usage of the program.

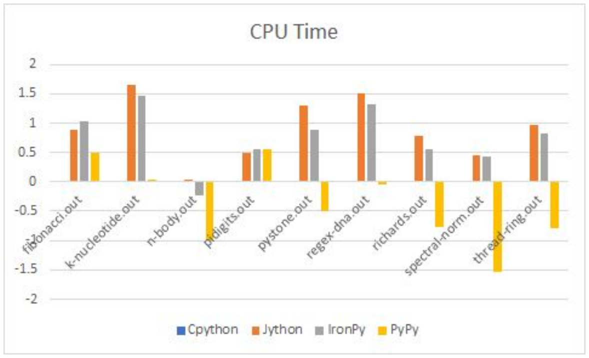
For CPU load, we measure the User time and Idle time before forking the child process and after the child process exits. And calculate the User time's fraction of total user time+total Idle time.

1.3.2 Subprocess

We meet some problem when trying to use the Psutil package on Jython and IronPython. These interpreters do not support the package. So we turned to use subprocess module, this allow us to call a subprocess from a Python script and we can also use the Psutil package to trace the subprocess as long as we have the process id of the subprocess, and Psutil can be used in a Python script with CPython Interpreter.

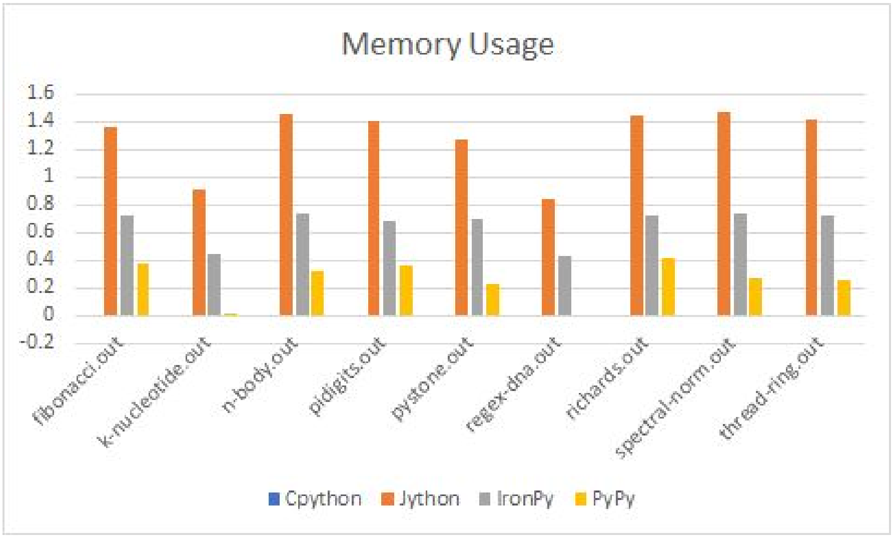
2. Measurement Result on Windows



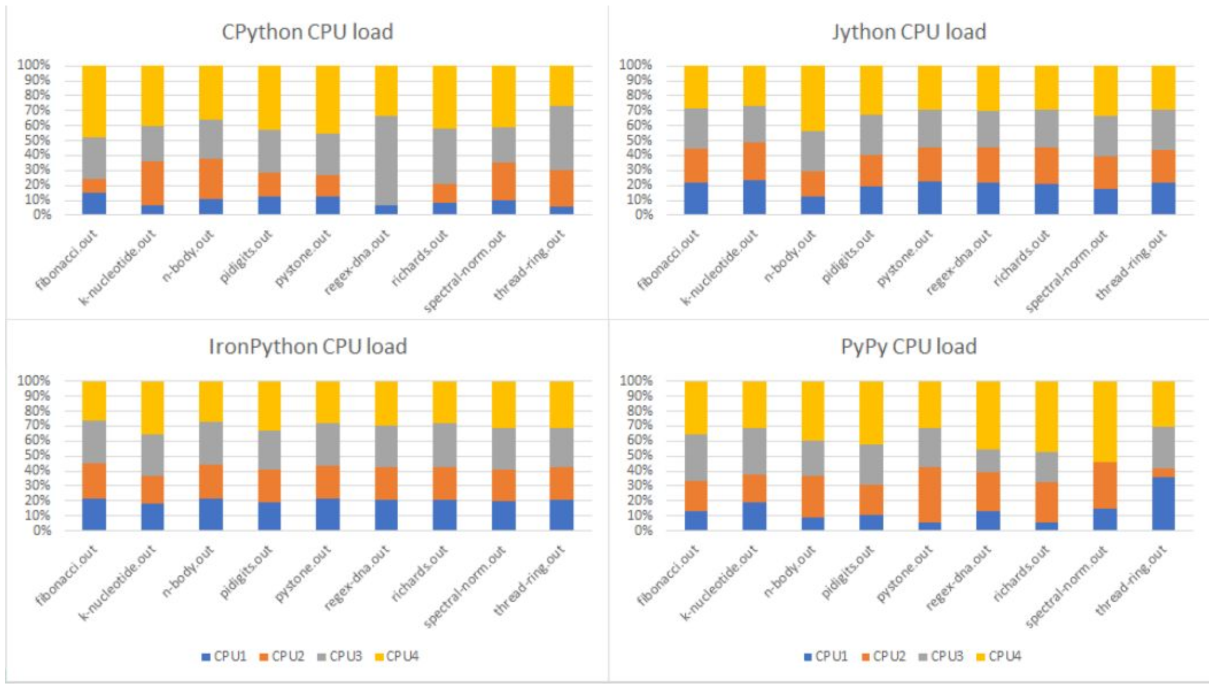


We pick CPython result as the reference because CPython is the most popular python Interpreter. we take log(Other interpreter's result / CPython result) to plot the picture above. PyPy has the highest speedup for most of the programs, however, IronPython and Jython slow down the program compared to CPython. A possible reason may be PyPy has JIT compiler that precompile the hot loop to save the time. However, for programs that are recursion, such as fibonacci.py, the JIT cannot speed up the precompile stage. For Jython and IronPython, the interpreter has to compile the python code to java and C# bytecode, therefore increase the elapsed time.

CPU time has the similar tendency as Elapsed time. and the reason are almost the same. we further discuss them in the conclusion.



Both Jython, IronPython and PyPy needs more memory spaces compared to Cpython. Jython and IronPython require more memory is because they have to call JVM and CLR to execute the bytecode they compiled. PyPy also needs more memory because of its JIT compiler and garbage collector.

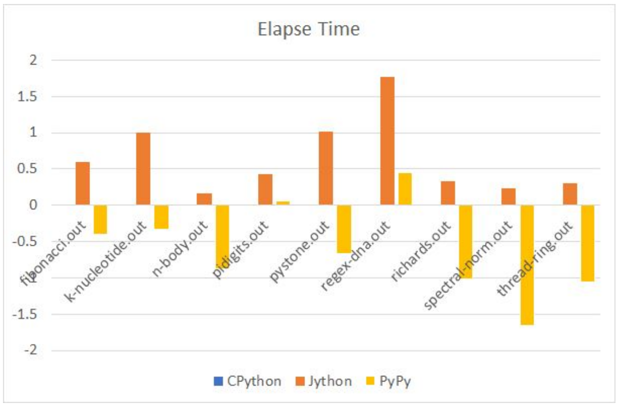


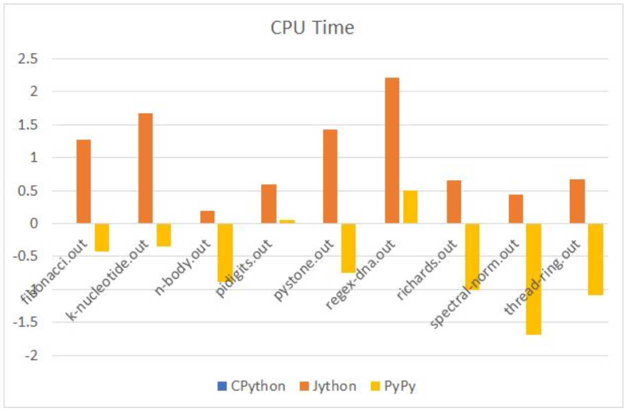
We measure the CPU load for each Interpreters compare to CPython, we find that Jython and IronPython allocate the job evenly on four cpus. So we measure the performance of parallelization for Jython, IronPython compared to CPython. We allocate a job on three interpreters with both single-thread and multi-thread. The result below shows that Jython and IronPython perform well in Multi-thread task, the Elapsed is decreased and the cpu load is almost 100% for all the cpus. However, multi-thread does not improve the performance of CPython, it is even worse compared to single-thread. The GIL in the CPython block the free thread from getting access to the resource.

Thread	Interpreter	Elapsed time	CPU time	CPU1 load/%	CPU2 load/%	CPU3 load/%	CPU4 load/%
Single	CPython	44.61	42.89	28	37	47	86
Multi		56.77	53.20	37	42	65	87
Single	Jython	39.51	43.89	43	51	76	82
Multi		13.41	43.56	90	90	93	95
Single	IronPy	81.86	81.95	19	30	61	86
Multi		58.17	152.15	82	85	90	90

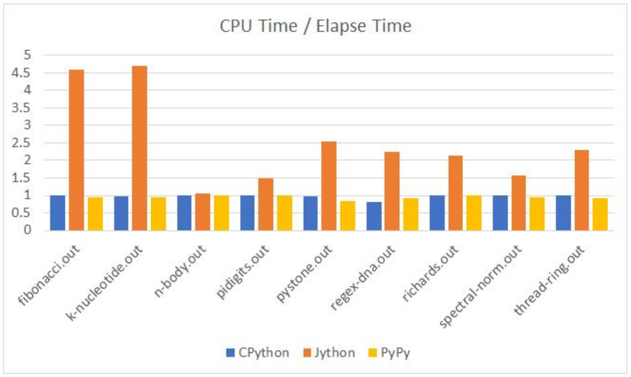
3. Measurement Result on Linux

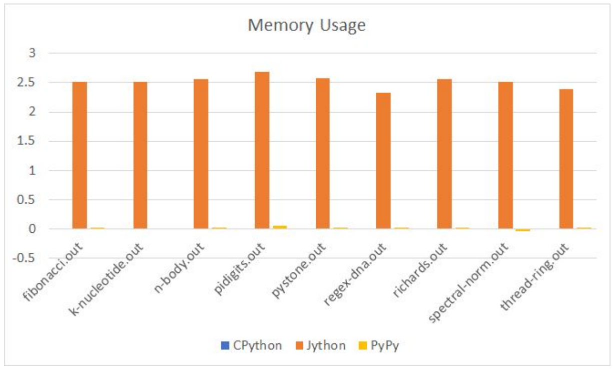
Similar to what we did in Windows system, we use psutil package and subprocess to measure the performance on Linux system(Ubuntu 16.04 LTS). Since we did not find a way to install IronPython on Ubuntu, here we skip IronPython, and only make comparison between Cpython, Pypy and Jython.
Taking Cpython as the reference, Pypy has speedup on 7 out of 9 programs(3~50 times faster), however there's 2 programs slower than CPython. All test programs with Jython interpreter are 2~50 times slower than CPython.
As for CPU time, the result has the same tendency as the elapsed time: Pypy has speedup for the most of the time, Jython has slowdown for all the programs.



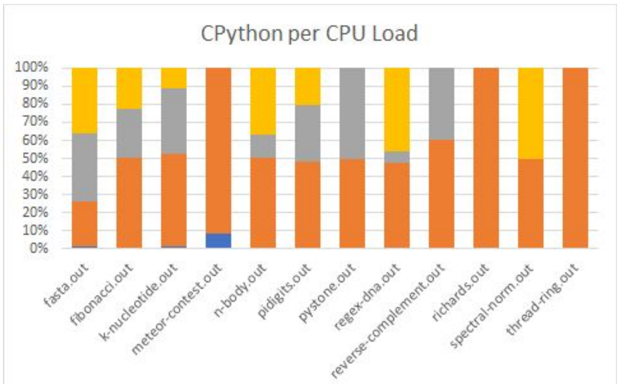


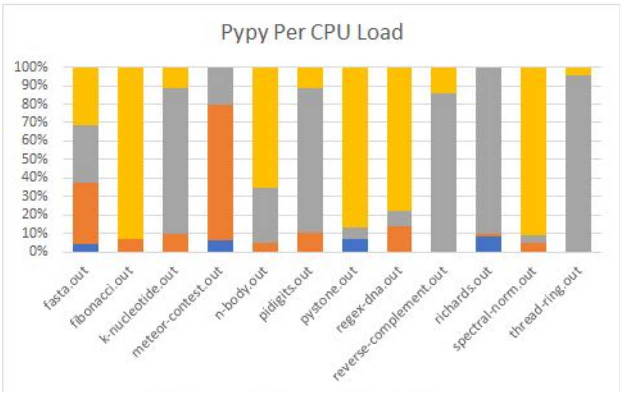
If we divide Total CPU Time by Elapsed time, then Jython has higher value among the three interpreters. Since we have 4 cores and the Total CPU time refers to the sum of the execution time on each core, then the higher the ratio means the more effective multi-threading. Jython has an average of 2.5 cores assigned for the program during the whole elapsed time. When it comes to CPython and Pypy, the average result is 1 core.

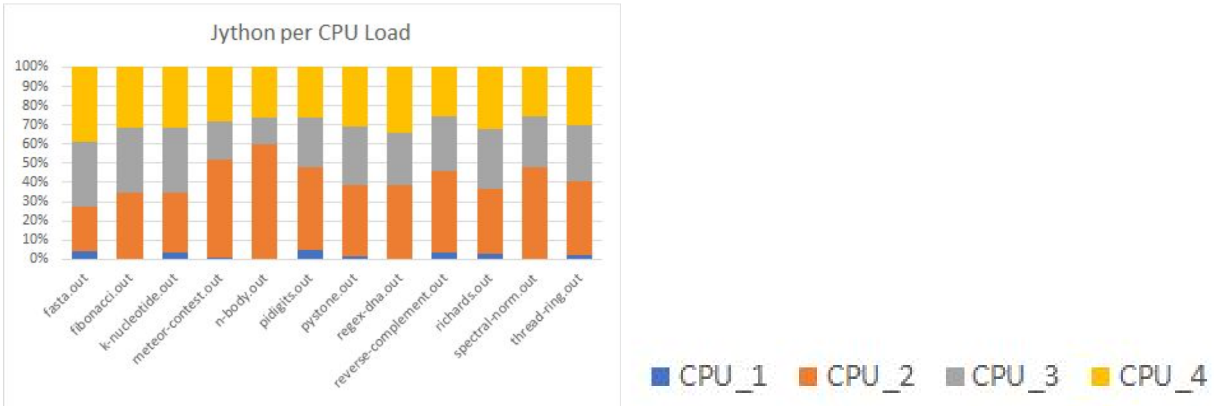




According to the memory usage figure, Jython allocates more memory during the runtime. Jython requires about 200~300MB memory because JVM will be called afterwards. Pypy and CPython has almost the same memory usage, which is a different result from what we got on Windows platform. On windows platform, pypy requires 70 percent more memory than CPython does.







The above figure shows the CPU load for each interpreters. CPython and Pypy tend to distribute tasks to 4 cores unevenly, while Jython will allocate tasks more evenly. It seems that Jython is better at utilizing multiple cores to do the computation. To verify this suppose, single-threaded and multi-threaded code are tested with these interpreters. The result shows that both Jython and Pypy well benefit from multi-threading, and will gain descent speedup from it. CPython takes longer time to execute multi-threaded code, and this is similar to the Windows result.

Thread	Interpreter	Elapsed time	CPU time	CPU1 load/%	CPU2 load/%	CPU3 load/%	CPU4 load/%
Single	CPython	0.56	0.55	51	19	98	1
Multi		0.57	0.56	3	100	77	26
Single	Jython	6.64	11.11	70	98	43	54
Multi		5.98	14.75	75	95	91	79
Single	Pypy	0.43	0.4	21	100	12	15
Multi		0.17	0.16	100	100	6	5

4. Observations and Analyzation

CPython is taken as the reference in our performance comparison experiments. It is the most popular pythonic interpreter because it is well supported. It does not require too much memory space executing programs(when compared to Jython and IronPython), so this is good for those memory constrained devices. In our experiments, CPython performs worse on multi-threaded tasks, because GIT (Global Interpreter Lock) will act as a mutex between the threads, to avoid conflicts and ensure thread-safe.

Pypy delivers stunning performance when compared with CPython. Thanks to its JIT feature, Pypy runs faster than CPython on most of the programs, with no memory overhead on Linux and little overhead on Windows. Its Just In Time compiler will compile some hot sections of bytecode into machine code, and that is where the speedup come from. However,not all programs will run faster than CPython. In our experiment, program fibonacci and pidigits even have longer elapsed time on Windows(program pidigits and regex-dna on Linux). We noticed that fibonacci and pidigits heavily reply on recursion and it seems that JIT is not good at dealing with it. We cannot conclude that this is the exact reason, because program regex-dna has no recursion in its code but still runs slower.

On both Windows and Linux platform, Jython and IronPython allocate more memory than Cpython and Pypy do. Moreover, they are significantly slower than CPython, not to mention Pypy. This is because everytime we execute a program with these two interpreters, they will start their own runtime environment(JVM for Jython, CLR for IronPython). And this require more memory to be allocated and longer time to execute. For those small size programs, the performance will be largely impacted by this introduced overhead. However, we expect these two interpreters could deliver better performance when dealing with larger programs, because they both have better multi-thread support, and that is exactly the drawback of the CPython.