OpenMP for Python Technical Overview

Caleb Huck

June 3rd, 2021 ************ change

1. OVERVIEW

OpenMP for Python is currently a prototype level software that brings core OpenMP function and behavior to the Python language. It is designated as a prototype because the goal of this project, to this point, was to implement a proof-of-concept that not only could OpenMP be adapted to the Python syntax and paradigm in a way that is intuitive and usable, but that parallel concepts (e.g. concurrency, speedup, efficiency, etc.) could be demonstrated clearly for the specific purpose of teaching. While performance is something we care about, in the context of demonstrating the benefits of parallelism, it is not our goal to provide a real world performance tool, as real world performance applications would always benefit from using a more appropriate language. To that end, we have kept the design as simple as possible in order to quickly realize our proof-of-concept and provide a base that can be continually developed in the future into a robust tool for teaching.

This software currently includes a specific subset of the full OpenMP specification. We have deliberately selected specific directives, clauses, and runtime functions that represent the core functionality needed to implement most any parallel algorithm. This subset of the full standard is sufficient for our needs but can easily be added to in the future if needed. It is important to note that, while it was our goal for the current implementation to follow the OpenMP standard as closely as possible at this time, it is still not "one-to-one" with OpenMP. There are likely still corner cases where the behavior of our software will differ with OpenMP. We have made an effort to test each construct during development to catch as many inconsistencies as possible, however, it will take significant user testing and iterative improvement in the future to fully capture OpenMP behavior, at least to the degree that it is possible. Additionally, there are some differences between our software and original OpenMP that are a result of the inherent differences between C-like languages and Python. This is inevitable given the many paradigmatic incongruencies between these languages.

This document provides a general overview of the design, architecture, and implementation of this software. Our goal is to provide background on the technical aspects of the project, explain the functional design and software architecture, and finally, demonstrate results from our microbenchmarks and cover some of the interesting points regarding them. Finally, the user API will not be included in this document. Instead, it will be provided in a separate user document. The organization of this document is as follows: the next section will give a high-level functional overview of the software architecture and its components, followed by a brief explanation of how data flows through the application.

2. PYTHON OBSTACLES

Early on in the design, we faced several problems that needed to be solved for our proof-of-concept to be possible. The first was the CPython GIL (Global Interpreter Lock). The GIL essentially makes it impossible to write truly parallel multi-threaded code by locking the interpreter every time a thread accesses any shared resource, effectively serializing the code and making it unlikely to gain any speedup through multithreading parallelism. We solve this issue by using Jython for our interpreter. The Jython interpreter is written in Java and based on the Python 2.7 standard (although some Python 3 features are also incorporated, such as *range()*). Jython uses Java threads under the hood and does not implement a GIL, allowing for truly parallel execution.

Another issue we faced had to do with data structures in Python. Python does not have primitive arrays like in C. In C, an array is just a pointer to the first item in the array. Python instead has *lists*. Lists are objects that wrap data collections and behave similarly to arrays syntactically (e.g., allowing bracket notation accesses). The problem is that lists are completely synchronized for both reading and writing, meaning that only one thread can access the list at a time, even if they are operating at different indices. We found a partial solution for this problem. For parallel data reading, it is possible to use jarray.array or jarray.zeros. These objects have almost identical member functions to lists, however, they are implemented differently under the hood using Java arrays and allow for parallel reading but not writing.

Some of the issues we ran into were the result of differences between Python and C-like languages, and ultimately came down to design decisions on how we wanted to handle them. For example, Python has no method for defining an arbitrary scope (which can be done in C by wrapping the code in curly braces, for example: { */*code with new scope*/* }). In C, this is how the preprocessor knows what code to apply the preceding OpenMP directive to. We solve this problem by modifying the Python syntax to allow our OpenMP comments to come before an indented block, defining a scope local to that directive. This has the unavoidable effect of breaking compatibility with standard Python if the code is not transformed by our preprocessor first, since it will appear to have a normal Python comment followed by an indented block, which is illegal in Python. We made the deliberate decision to sacrifice compatibility with standard Python interpreters for the sake of keeping consistent with Python indentation convention and to have the most intuitive mapping from C OpenMP to Python as possible.

3. ARCHITECTURE OVERVIEW

The basic flow of the application is shown in figure 16. The large arrows represent the flow of the source code through the software and the small arrows denote communication that doesn't involve the source code. We use a modified version of the Jython interpreter since it does not implement a Global Interpreter Lock like CPython and therefore is able to run multithreaded code truly in parallel, using the underlying Java threads (Jython is written in Java). Currently, our OpenMP source-to-source preprocessor is written in Python. The reason for this is that Python was an easy choice early on for quick prototyping and testing Python code transformation before we had made a commitment in terms of design and project direction. If we had been sure about our approach using Jython from the beginning, we likely would have written the preprocessor in Java in order to have a more seamless integration between interpreter and preprocessor. Nonetheless, our current, more separated approach allows for flexibility in the future if we want to use our preprocessor with a different interpreter. Additionally, porting our solution to Java would also be a straightforward task if desired since our parser generator (ANTLR4) can easily be used with Python or Java.

Since our preprocessor is written in Python, it runs in a separate process than the interpreter. We accomplish this by spawing a subprocess using the Java ProcessBuilder class. This allows us to start the preprocessor, passing in the fully qualified path to the source code file as a command line argument to the python process, and finally capturing the output (whether it be the transformed source code, or an error output).

The preprocessor consists of two main components: the ANTLR lexer/parser (which are automatically generated by ANTLR from our modified Python grammar file), and the Translator Visitor. The Translator Visitor traverses the abstract syntax tree produced by the parser and is responsible for creating the multithreaded code that will run on Jython. The visitor object contains one visitor function per rule from the grammar that is triggered every time that rule is encountered. If it is rule pertaining to regular Python statements, then the visitor just "prints" the code with no changes. But if the rule pertains to an OpenMP statement, then the visitor function instead prints the multithreaded equivalent of the OpenMP code. Exactly how the code is transformed will be covered in more detail later. At this point, if no errors were generated during the lexing/parsing, the new source code is printed to stdout, where it is then captured by Jython and written to a temporary file. In contrast, if an error was generated, only the error is printed and will be displayed to the user from Jython before exiting with an error status.

Lastly, some of the output code, such as OpenMP for loops, require runtime calls to be made. For example, if the schedule is dynamic, then each thread must make a call to request the next block each time it finishes an assigned block. This is accomplished using the backend runtime, which is not user-visible. In contrast, the omp runtime contains OpenMP API function calls (e.g. `omp_get_thread_num()` and `omp_get_num_threads()`), and is user visible. Both of these modules reside in a directory within the preprocessor file structure which is added to the path automatically during code transformation. check how user imports ompy after benchark finishes.

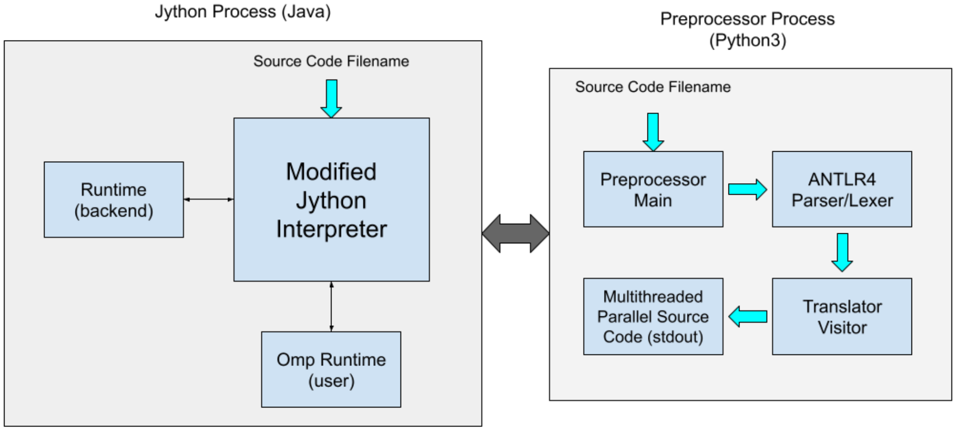

Figure 1: Architecture Overview

4. CODE TRANSFORMATION

This section gives insight into how the code is transformed from OpenMP blocks to multithreaded code and the strategy behind the preprocessor. Below are some examples of the primary OpenMP constructs and how they are translated into multithreaded code that Jython can run. Note that some of the "boilerplace" imports and code for setting the correct path have been left out of the transformed code for the sake of being concise. Also note that all variables beginning and ending with a "_" are variables created by the preprocessor and use this convention to avoid naming collisions with user variables. Therefore, we require that user variables not begin and end with a "_".

4.1. Parallel Construct

A simple example of how a parallel directive is translated is given in Figures 2 & 3. There are several important things here to note. First, since there is no `num_threads()` clause given, the `_num_threads_` variable is set to the output of `os.cpucount()` by default, which in the case of the computer this was run on, was 8. The parallel region is wrapped in a function, which is passed to the runtime.submit function to execute the parallel region with the correct number of threads. Every parallel target function takes a `RuntimeManager` object as an argument. The manager is not needed in this particular example but will be shown in later examples. Finally, the way in which we handle variable scoping for private and shared are both demonstrated. We use the global keyword to tell Python we want to use the closure feature to bind the outer var1 to the inner function scope. Next we use the copy-on-write rule of Python to create a new copy of var2 that is local to the function scope and has the initial value of 0.

Figure 2: Parallel Constuct - Original Code

```
var1 = 'var1'
var2 = 'var2'
#omp parallel shared(var1) private(var2)
    print(var1)
    print(var2)
```

Figure 3: Parallel Construct - Transformed Code

```
_num_threads_ = 1
var1 = 'var1'
var2 = 'var2'
_num_threads_ = 8
def _target_0(_manager_):
    global var1
    var2 = 0
    print var1
    print var2
_manager_outer_ = RuntimeManager(_num_threads_)
submit(_target_0, _num_threads_, args=(_manager_outer_,))
```

4.2. For Construct

Due to the fact that Python only supports for each loops, we had to enforce some restrictions on how they can be used with OpenMP for directives. At the parser level, we enforce that the structure of a for loop following a for directive must be of the form "`for [single variable] in range([1, 2, or 3 parameters]):`". If one parameter (`n`) is passed to `range()`, then the returned range will be `[0, n-1]`. If two parameters are passed (`k`, and `n`), then the range will be `[k, n-1]`. Finally, if three parameters are passed (`k`, `n`, and `i`), then the range will be `[k, k+i, k+2i, k+3i, ..., n-1]`. This ensures that OpenMP for loops will behave as close to C-like for-loops as possible. Figures 4 & 5 show how an OpenMP for block are translated inside of a parallel region. The if statement at line 8 ensures that only thread 0 creates the for loop manager object that will partition the work among the threads. The `set_for()` method on line 10 makes the manager available to all threads through the shared manager object. Finally, the structure of the actual for construct is an infinite loop (line 17) that continually requests the next block to execute until the for loop manager `done()` method returns `True` (line 23). The manager dispatches blocks of work according to the schedule (which can be static, dynamic, or guided). Additionally, the critical directive in the for loop is implemented by surrounding the critical section in a lock that all threads have access to through the manager.

Figure 4: Parallel Constuct - Original Code

```
sum = 0
#omp parallel num_threads(2) shared(sum)
    #omp for schedule(dynamic, 5)
        for i in range(0, 100, 2):
            #omp critical
                sum += 1
```

Figure 5: Parallel Construct - Transformed Code

```
_num_threads_ = 1
sum = 0
_num_threads_ = 2
def _target_0(_manager_):
    global sum
    _schedule_ = 'dynamic'
    _chunk_ = 5
    if omp_get_thread_num() == 0:
        _for_manager_ = ForManager(_schedule_, _chunk_, _num_threads_)
        _manager_.set_for(_for_manager_)
    else:
        _for_manager_ = _manager_.get_for()
    _arg1_ = 0
    _arg2_ = 100
    _arg3_ = 2
    _for_manager_.setup(_arg1_, _arg2_, _arg3_)
    while True:
        _start_, _stop_, _step_ = _for_manager_.request()
        for i in range(_start_, _stop_, _step_):
            _manager_.critical_lock.acquire()
            sum += 1
            _manager_.critical_lock.release()
        if _for_manager_.done(): break
_manager_outer_ = RuntimeManager(_num_threads_)
submit(_target_0, _num_threads_, args=(_manager_outer_,))
print sum
```

4.3. Reduction Construct

Figures 6 & 7 show not only reduction, but also how parallel and for can be combined into a single line with the same effect as if they are separate. The reduction works by first creating a private variable using the copy-on-write rule (line 7) the same as with the `private()` clause from the earlier example. At the end of the parallel region, there is a call to the `update_reduction_variable()` method of the manager (line 17). This saves a copy of the last value of the reduction variable for each thread in the manager object. After the parallel region is executed, the value of the outer version of the reduction variable is updated with a call to the `get_reduction_value()` method of the manager, which aggregates the values from each thread according the user specified aggregation operation.

Figure 6: Reduction Construct - Original Code

```
sum = 0
#omp parallel for reduction(+:sum)
    for i in range(10):
        sum += 1
print(sum)
```

Figure 7: Reduction Construct - Transformed Code

```
_num_threads_ = 1
sum = 0
_num_threads_ = 8
_schedule_ = None
_chunk_ = None
def _target_0(_manager_):
    sum = 0
    _arg1_ = 10
    _arg2_ = None
    _arg3_ = None
    _for_manager_.setup(_arg1_, _arg2_, _arg3_)
    while True:
        _start_, _stop_, _step_ = _for_manager_.request()
        if _start_ == _stop_: break
        for i in range(_start_, _stop_, _step_):
            sum += 1
    _manager_.update_reduction_variable(’sum’, sum, ’+’)
_for_manager_ = ForManager(_schedule_, _chunk_, _num_threads_)
_manager_ = RuntimeManager(_num_threads_)
_manager_.set_for(_for_manager_)
submit(_target_0, _num_threads_, args=(_manager_,))
sum = _manager_.get_reduction_value(’sum’)
print sum
```

5. BENCHMARKS

To test the performance of our application, we run three micro-bencmarks featuring simple, well-known parallel algorithms. The first is nxn matrix multiplication. Normally, this would be an “embarrassingly parallel” problem, however, because Python doesn’t have truly parallel arrays, this isn’t completely true. The matrices are stored in jarray.array objects and we use row-wise partitioning to divide the work.

The next micro-benchmark is parallel sum. Similarly, the integers to be added are stored in a jarray.array object, which is divided evenly among the threads.

The last micro-benchmark is parallel bubble sort. finish this after handling openmp threads only vs serial algo

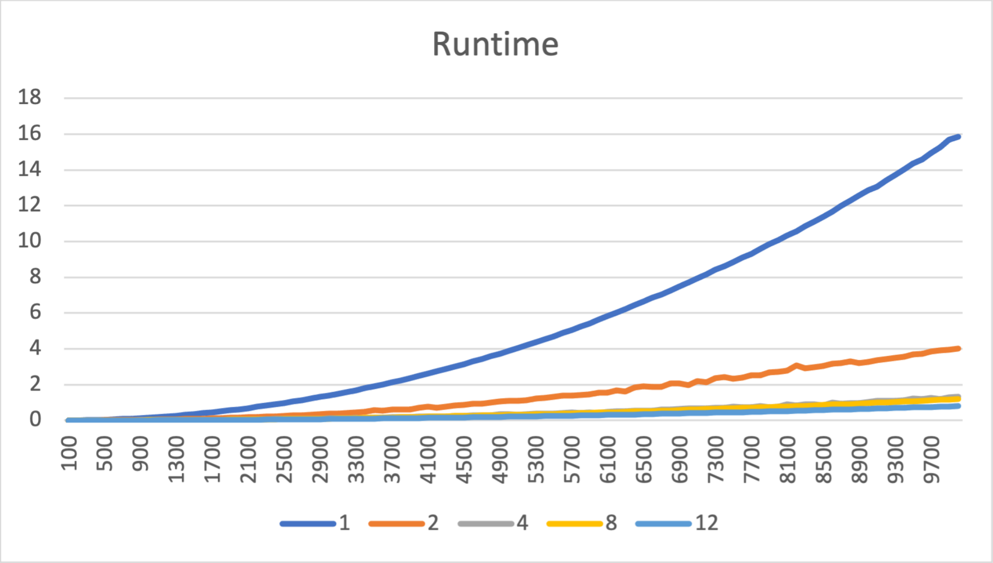

Figure 8: Parallel Bubble Sort - Runtime

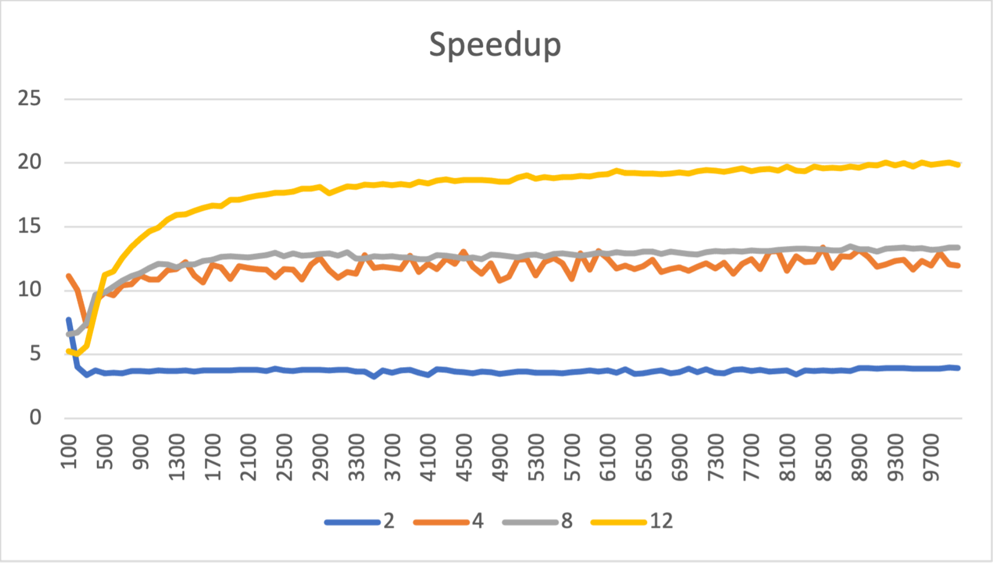

Figure 9: Parallel Bubble Sort - Speedup

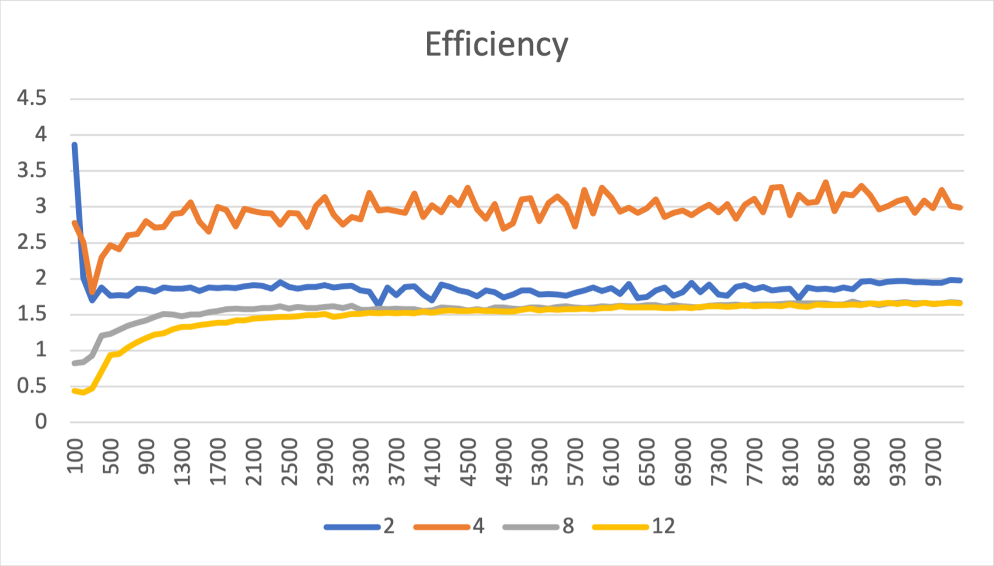

Figure 10: Parallel Bubble Sort - Efficiency

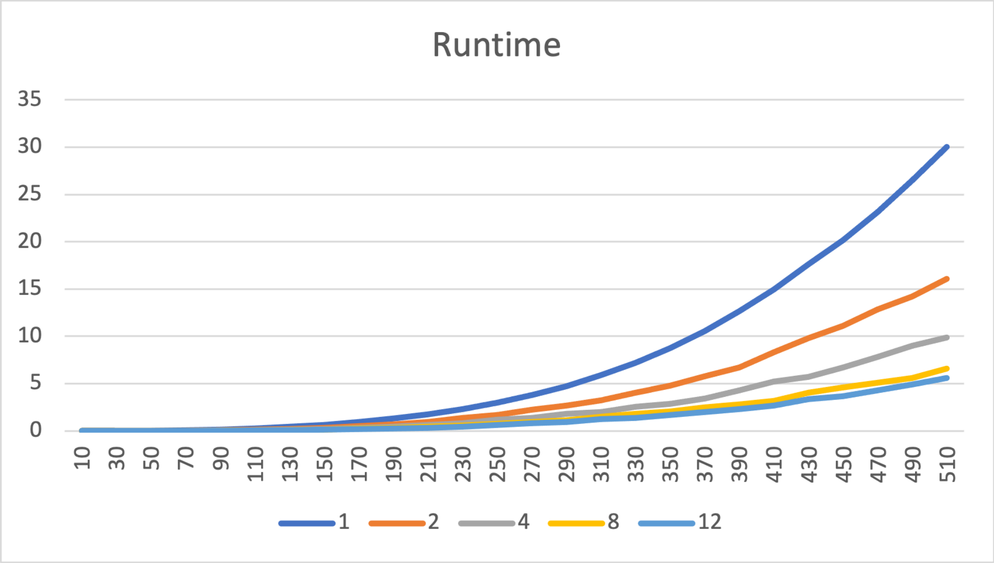

Figure 11: Matrix Multiplication - Runtime

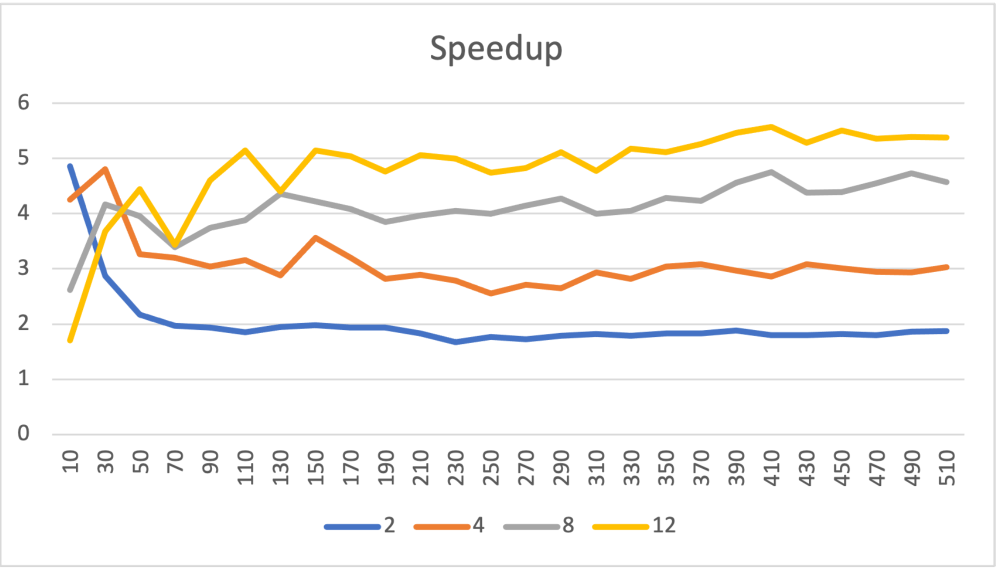

Figure 12: Matrix Multiplication - Speedup

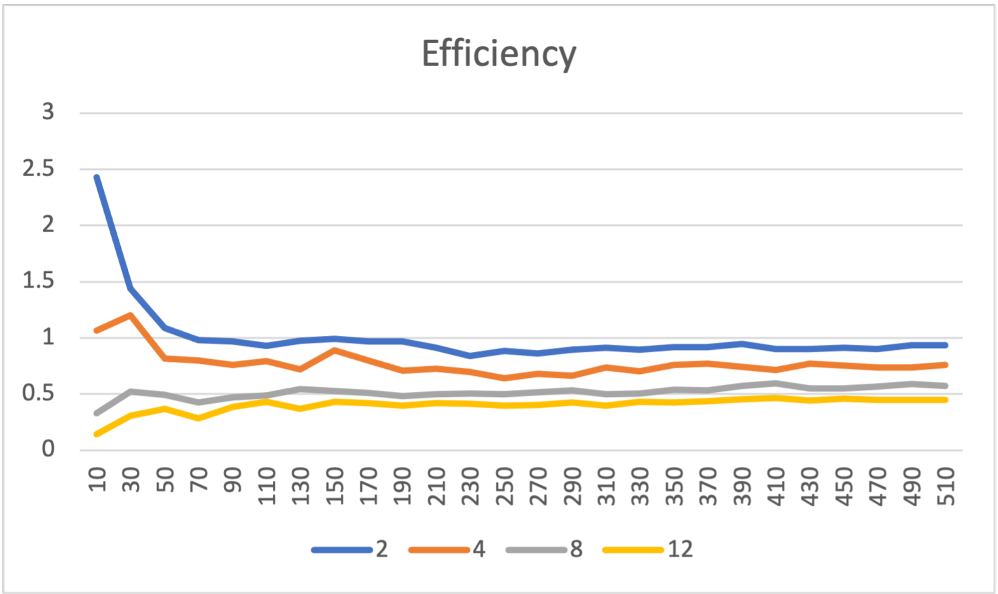

Figure 13: Matrix Multiplication - Efficiency

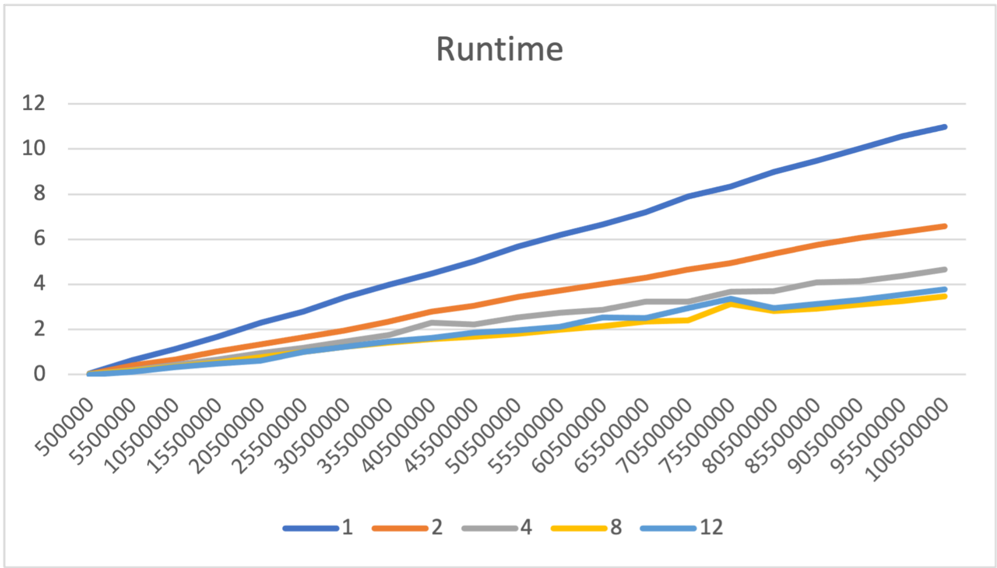

Figure 14: Parallel Sum - Runtime

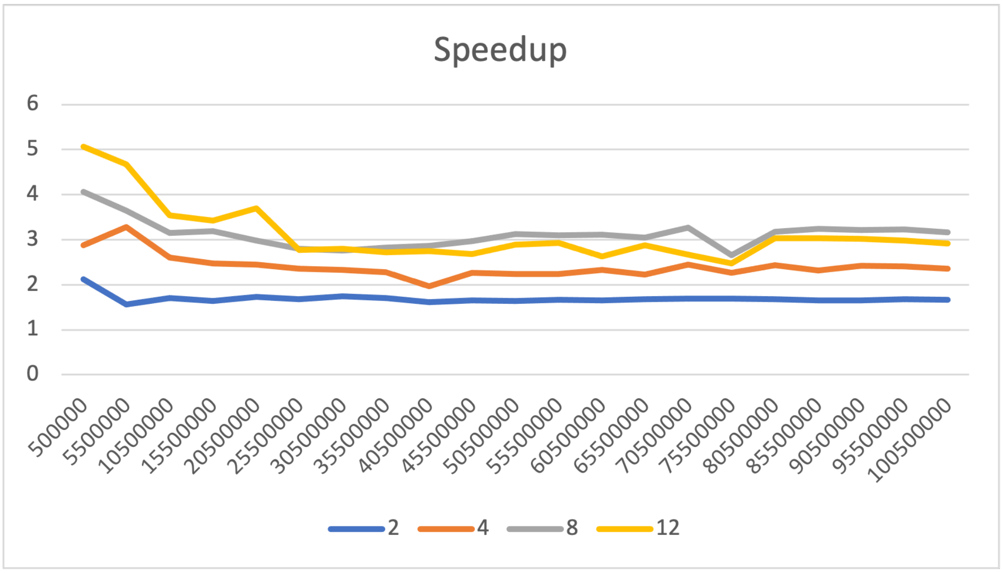

Figure 15: Parallel Sum - Speedup

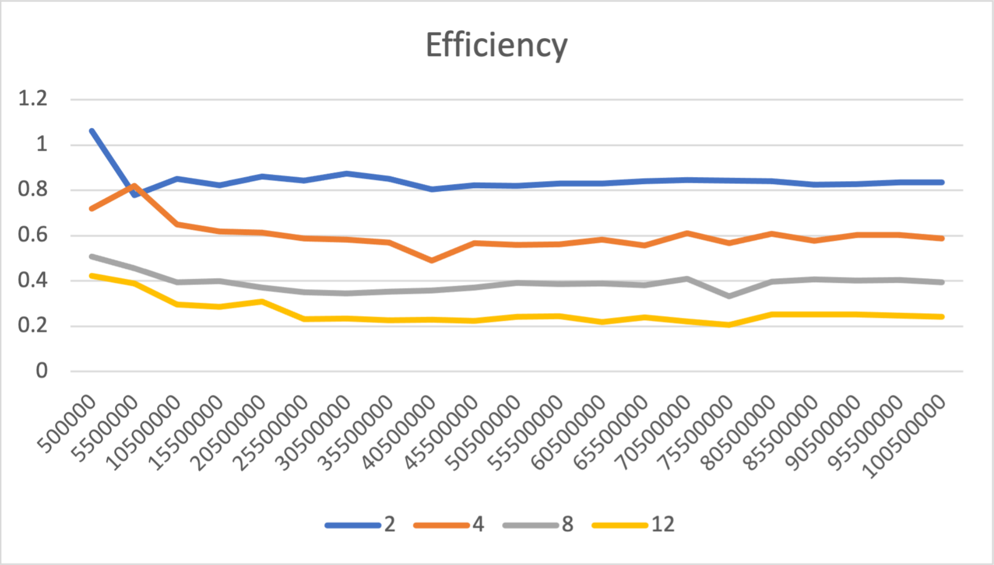

Figure 16: Parallel Sum - Efficiency