OpenMP for Python Technical Overview

Caleb Huck

June 8th, 2021

1. Overview

In recent years, multi-core systems have become ubiquitous in computing. From supercomputers, all the way down to budget laptops and cell phones, leveraging multiple processors has become the norm for essentially every computing device. With single-core systems becoming a thing of the past, parallel and distributed programming skills have become more important and relevant than ever and are only becoming more so. Naturally, this has led to increasing pressure in the academic space to incorporate and integrate these topics into CS curricula. Traditionally, parallel/distributed topics have been considered an HPC (High-Performance Computing) topic and were reserved for higher-level elective courses, being left out of mainstream curricula almost entirely [1]. However, the landscape is rapidly changing, and the number of jobs requiring knowledge in this area has increased drastically. For this reason, there has been significant work done in the literature towards integrating these topics into early CS curricula [2], [3], [4], [5], [6], [7], [8], [1], [9], [10], [11], [12].

iPDC (Integrating Parallel and Distributed Computing) [13] is an initiative at Tennessee Technological University aimed at providing education and resources to professors who teach early CS courses to assist them in integrating parallel and distributed topics into their curricula. Part of the program is a week-long workshop for professors that includes lectures and hands-on programming assignments using several different programming languages and tools. One of these is OpenMP. OpenMP (Open Multi-Processing) is a library for FORTRAN, C, and C++ that provides a high-level API for parallel, shared-memory programming. The primary advantage of OpenMP is that it removes responsibility for thread creation and management from the programmer. Instead, the API consists of abstract, preprocessor level directives and clauses that let the programmer describe how the code should run and what concurrent dependencies exist without writing the threaded code themselves. Making parallel code easier to write allows students to focus on the concepts without being overwhelmed by the difficulty of writing low-level threaded code. This makes OpenMP an ideal resource for teaching these concepts to early CS students.

Python is now used in roughly one third of universities for teaching early CS courses, which was the primary motivation behind developing OpenMP for Python. OpenMP for Python is currently a prototype level software that brings core OpenMP function and behavior to the Python language. The goal of this project, to this point, was to implement a proof-of-concept demonstrating that OpenMP could be adapted to the Python syntax and paradigm in a way that is intuitive and usable. Additionally we wanted to show that parallel concepts (e.g. concurrency, speedup, efficiency, etc.) could be demonstrated clearly for the purpose of teaching early CS students. 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. 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 further 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: first, we will cover works related to our project that we learned and gained insight from, then we will cover some of the Python-specific problems we faced with the design of this software and how we went about solving them. Next we will give a high-level architecture overview and explain the software components and how they are organized and work together. After that, we will go over how the preprocessor transforms the OpenMP code into multi-threaded Python and give some examples. Finally, We will present two microbenchmarks we used to test the performance of our solution and briefly discuss the results.

2. Related Work

As far as we are aware, there have only been three attempts in the past to port OpenMP functionality to other languages, all of which have focused on Java as the target language. This project is the first effort to bring OpenMP to Python. However, we have learned from these other projects and derive much of our design approach from what worked well with them and what didn't work. Therefore, we include them them here, along with a brief summary of each project.

Jomp [14] was the first effort to bring OpenMP functionality to Java. It was based on the first iteration of the OpenMP specification and therefore implemented only a subset of the full specification available today. The software consisted of a preprocessor for transforming Jomp code to standard Java code, as well as a runtime library to implement OpenMP functions. To use Jomp, the user would write their source code in a .jomp file. The OpenMP directives are added in the form of single line Java comments. The Jomp preprocessor would then take this file and convert it to the corresponding thread-based Java code. The Java files must then be compiled using javac in order to run the program.

Pyjama [15] was a project building on the work from Jomp. Their solution used a similar approach with a few notable differences. First, Pyjama combined the preprocessor and java compilation into one application, effectively eliminating one step from the process compared to Jomp. Second, Pyjama added GUI specific directives for releasing the main GUI thread to prevent graphical applications from freezing or becoming unresponsive when doing parallel computation in the background. Lastly, Jomp was released in the year 2000, 13 years before Pyjama. Jomp was based on OpenMP 1.0 and Java 1.1, whereas, by the time Pyjama was released, OpenMP 4.0 was already out along with Java version 7. Out of the three, Pyjama still has the best coverage of the updated OpenMP specification.

Omp4j [16] is the most recent attempt at bringing OpenMP to Java. It is written in Java and Scala and works much the same as Pyjama, with one significant difference. Omp4j does not include a runtime library. The reason for this was to eliminate any extra dependencies. Instead, they use macros to emulate the function of the `omp_get_thread_num()` and `omp_get_num_threads()` functions without making an actual function call.

3. 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 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 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 automatically locked 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 statements (which are given in the form of a Python comment that our preprocessor can recognize) 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.

4. Architecture Overview

The basic flow of the application is shown in figure 1. The large, bright blue arrows represent the flow of the source code through the software. As mentioned before, we use a modified version of the Jython interpreter since it does not implement a Global Interpreter Lock like CPython. 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 might have just written the preprocessor in Java in order to have a more seamless integration between interpreter and preprocessor. Nonetheless, our current, 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 operating system-level process than the interpreter. We accomplish this by spawning 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 a rule pertaining to regular Python statements, then the visitor simply "prints" the code with no changes. But if the rule is an OpenMP rule, then the visitor function translates the OpenMP code to equivalent multithreaded code. How the code is transformed will be covered in more detail later in the document. 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 and executed. 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 function calls, and is user visible.

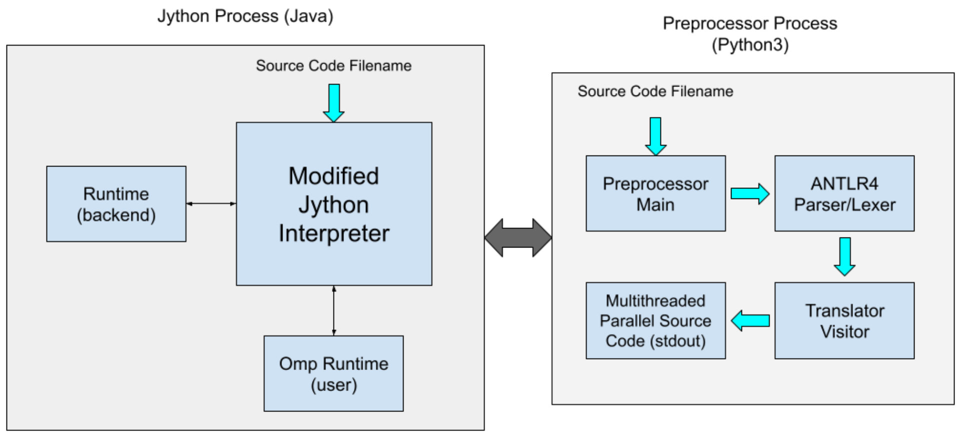

Figure 1: Architecture Overview

5. 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 "boilerplate" imports and code for setting the correct path have been left out of the examples 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 "_".

5.1. 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 when the threads need to communicate with each other or share objects. Finally, the way in which we handle variable scoping for the `private` and `shared` clauses 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_,))
```

5.2. 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 `ForManager` object's `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 (lines 20-22).

Figure 4: For 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: For 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
```

5.3. 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` clause 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 to the user specified aggregation operation.

Figure 6: Reduction Construct - Original Code

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

Figure 7: Reduction Construct - Transformed Code

```python
_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
```

6. Benchmarks

To test the performance of our application, we ran two micro-benchmarks featuring simple, well-known parallel algorithms. Both benchmarks were run on a Windows 10 computer with 6 cores/12 hardware threads and each data point is the average of 20 runs. The first benchmark is `nxn` matrix multiplication. The matrices are stored in `jarray.array` objects and consist of random integers. We use row-wise partitioning to divide the work. The runtime, speedup, and efficiency are shown in figures 8, 9, and 10 respectively.

The second micro-benchmark is parallel sum. Again, the integers to be added are stored in a `jarray.array` object, which is divided evenly among the threads. The results of this test are shown in figures 11, 12, and 13.

As mentioned previously, performance and scaling are not the primary goals of this project. We are not trying to make a tool for performance gains, but one for teaching parallel concepts. That being said, it is import to demonstrate the effects of parallelism for teaching, even if the results are suboptimal. The results we have obtained do this, albeit with some interesting, unexpected outcomes. The most obvious of these is the high speedup at the very beginning followed by a sharp drop off and then leveling out pattern that we observed across many of the runs. For matrix multiplication, the runs with 2 and 4 threads saw superlinear speedup at the smallest problem sizes and the the run with 2 threads for parallel sum did as well. We believe this has to do with the Just In Time (JIT) compilation in Jython, which adds significant overhead on the first execution to translate the Python code to Java byte code, which the later runs with more threads don't have to do. Additionally, while lower thread counts seem to perform very well in terms of efficiency, there seems to be quickly diminishing returns as the thread count goes higher. We believe this is because of the additional overhead of synchronization when writing to the `jarray.array` objects.

While our results require further investigation and testing to better understand the nuance of running on Jython and the underlying JVM, our results have met our initial goals of demonstrating parallelism and speedup. In the future, we plan to continue to refine our testing process to develop and improve the quality of our software.

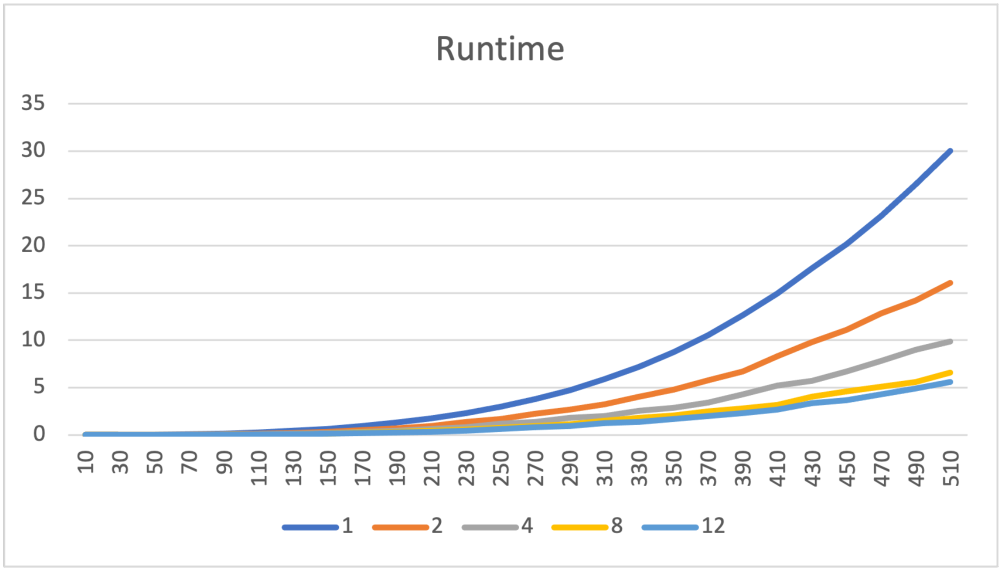

Figure 8: Parallel Matrix Multiplication - Runtime

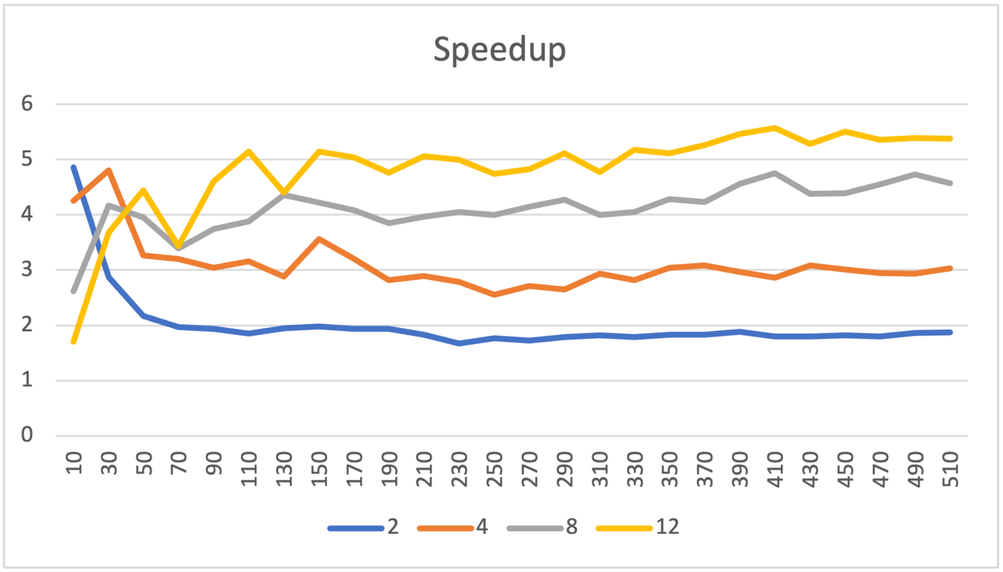

Figure 9: Parallel Matrix Multiplication - Speedup

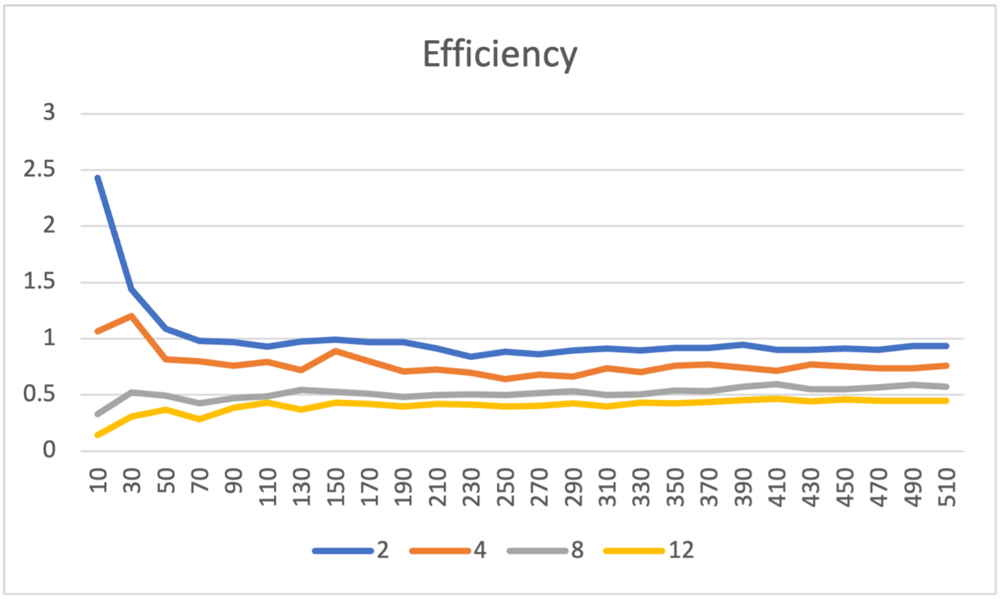

Figure 10: Parallel Matrix Multiplication - Efficiency

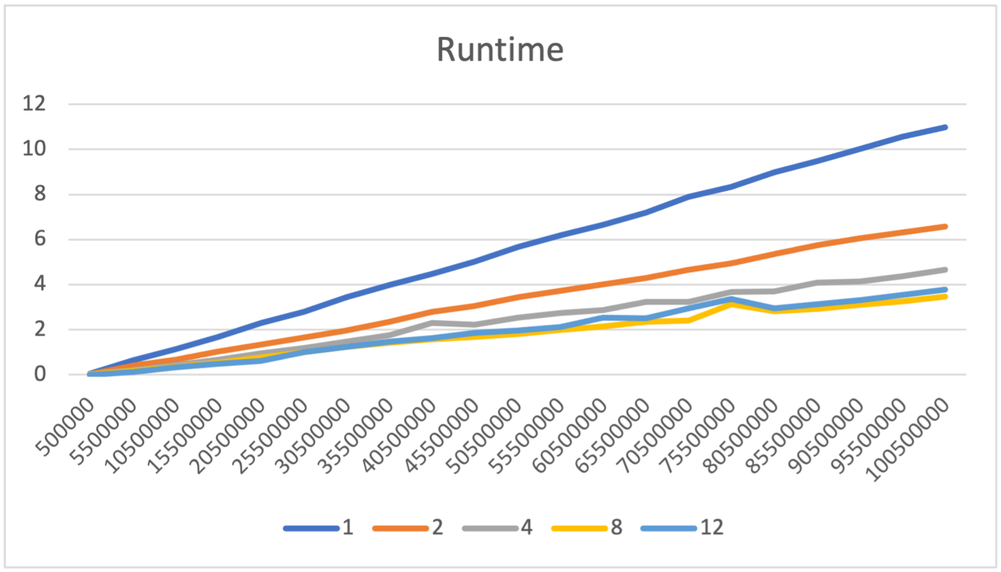

Figure 11: Parallel Sum - Runtime

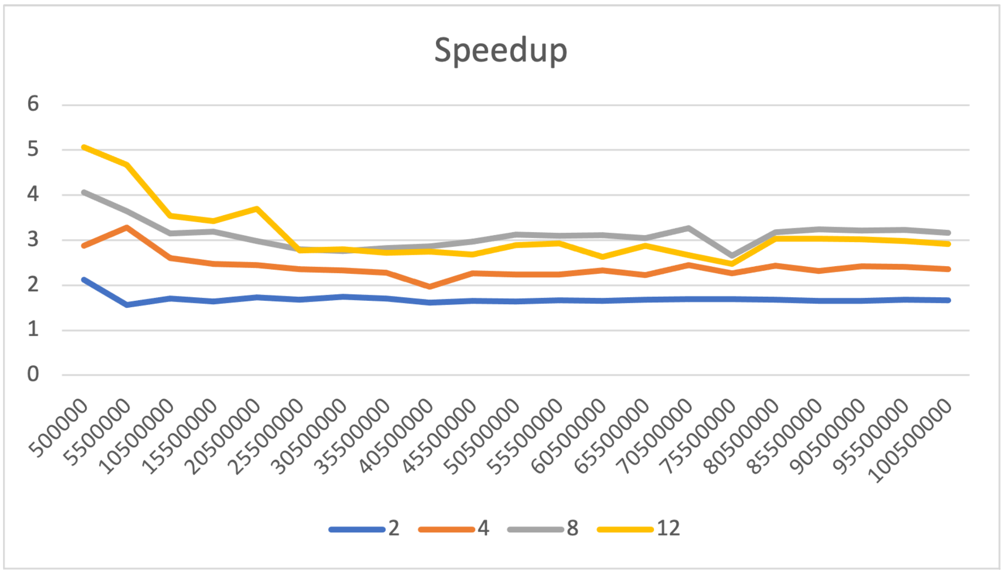

Figure 12: Parallel Sum - Speedup

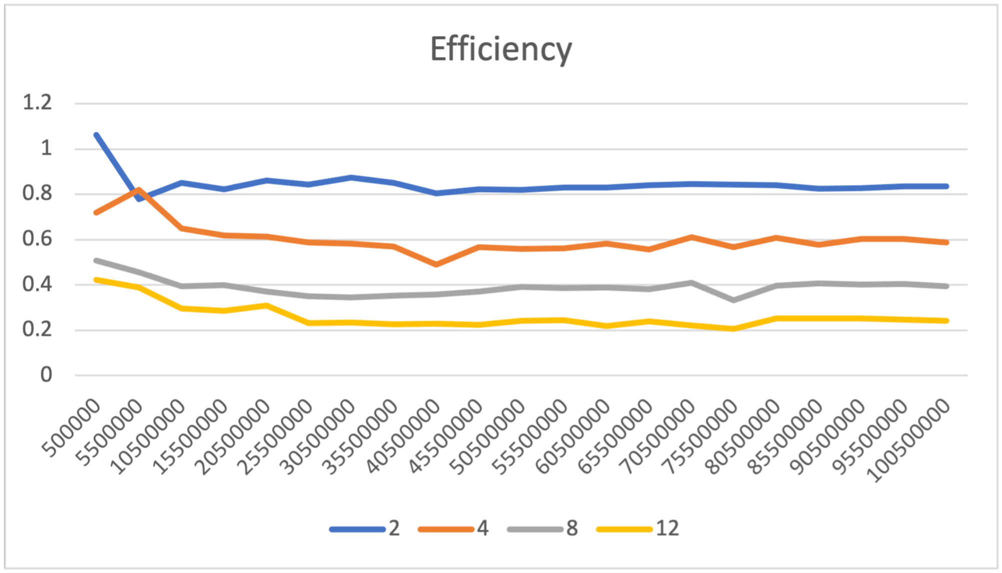

Figure 13: Parallel Sum - Efficiency

References

[1] D. J. Ernst and D. E. Stevenson, "Concurrent cs: preparing students for a multicore world," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pp. 230–234, 2008.

[2] Y. Ko, B. Burgstaller, and B. Scholz, "Parallel from the beginning: The case for multicore programming in thecomputer science undergraduate curriculum," in *Proceeding of the 44th ACM technical symposium on Computer science education*, pp. 415–420, 2013.

[3] J. C. Adams, "Injecting parallel computing into cs2," in *Proceedings of the 45th ACM technical symposium on Computer science education*, pp. 277–282, 2014.

[4] B. Neelima and J. Li, "Introducing high performance computing concepts into engineering undergraduate curriculum: a success story," in *Proceedings of the Workshop on Education for High-Performance Computing*, pp. 1–8, 2015.

[5] J. R. Graham, "Integrating parallel programming techniques into traditional computer science curricula," *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 75–78, 2007.

[6] R. Brown and E. Shoop, "Csinparallel and synergy for rapid incremental addition of pdc into cs curricula," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 1329–1334, IEEE, 2012.

[7] R. Brown and E. Shoop, "Modules in community: injecting more parallelism into computer science curricula," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 447–452, 2011.

[8] J. Adams, R. Brown, and E. Shoop, "Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to cs undergraduates," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1244–1251, IEEE, 2013.

[9] W. Ahmed, M. M. Muthaher, and J. M. Basheer, "Introducing high performance computing (hpc) concepts in institutions with an absence of hpc culture," in *2013 Sixth International Conference on Contemporary Computing (IC3)*, pp. 274–277, IEEE, 2013.

[10] S. A. Bogaerts, "Limited time and experience: Parallelism in cs1," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pp. 1071–1078, IEEE, 2014.

[11] B. L. Kurtz, C. Kim, and J. Alsabbagh, "Parallel computing in the undergraduate curriculum," in *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pp. 212–216, 1998.

[12] G. Lu, J. Xu, J. Liu, B. Dai, S. Gui, and S. Zhan, "Integrating parallel and distributed computing topics into an undergraduate cs curriculum at uestc," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, (Los Alamitos, CA, USA), pp. 782–787, IEEE Computer Society, may 2015.

[13] S. Ghafoor and M. Rogers, "ipdc workshop." `https://www.csc.tntech.edu/pdcincs/`.

[14] J. M. Bull and M. E. Kambites, "Jomp—an openmp-like interface for java," in *Proceedings of the ACM 2000 conference on Java Grande*, pp. 44–53, 2000.

[15] N. Giacaman and O. Sinnen, "Pyjama: Openmp-like implementation for java, with gui extensions," in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 43–52, 2013.

[16] P. Bělohlávek, "Openmp for java," 2015.