Lab Project Report[*]

Assignment 3

Christopher Esterhuyse, Mihai-Andrei Spadaru

1 Introduction

Multi-threaded programming is necessary to squeeze speed out of a program for a large problem. In a previous report, we explored the impact of multi-threading by way of OpenMP pragmas, leveraging task and data parallelism. As mentioned in that report, these OpenMP pragmas map to common, useful patterns that suit many problems. For some instances, the interface of compiler is not expressive enough, and the available OpenMP pragmas are not exhaustive enough to be best suited to a given problem. In such cases, threads are created irregularly, threads communicate in a complicated pattern, or indeed there exists some other idiosyncrasy of the input data that facilitates cutting corners (safely!) in a way too hopelessly involved or convoluted to express in a pragma. In such cases, it is time for the proverbial gloves to come off, and for the programmer to get down and dirty with manual control using POSIX Threads ('PThreads').

PThreads refers to an API put forward in the standard EEE Std 1003.1c-1995. The intent was to lay the groundwork for a standard means for one program to have several flows of control. Their 'lightness' brought with them great terseness and ease-of-use for the programmer, but necessitated some extra care for prickly race conditions that come as a consequence of a shared memory space. These concepts and the concurrency primitives that they necessitate are, in C, provided in library functions.

In the sections that follow, we outline some C implementations[1] for interesting problems, each of which is built on the use of PThreads. The execution times ('runtimes') of these programs are measured and interpreted as part of various experiments[2]. To follow, sections 2 to 4 focus on one such program each. The concise plots therein are complemented by more exhaustive tables in the Appendix.

2 Temperature Dissipation on the Surface of a Cylinder

The simulation of a cylindrical surface (represented as a 2-dimensional matrix *in silico*), was effectively parallelized using OpenMP. The predictably well-balanced workload lent itself well to the standard data-parallelism afforded by the OpenMP 'for' pragma. Can we achieve even greater speedup by really taking a hands-on approach to managing the threading overhead?

2.1 Implementation

Use of PThreads allows finer control of the mapping from one's algorithm to the what will end up running 'on the metal' and the details of each worker. Ultimately the high-level 'algorithmic' decisions are the same as in something like the OpenMP implementation, but now there are also new decisions to be made about how to manage thread handles, whether to spawn threads attached or detached etc.

[*]This report was submitted for the lab assignment of the 2018 edition of the Universiteit van Amsterdam Programming Multi-Core and Many-Core Systems master course.

[1]For uniformity, all C programs in this work were compiled with GCC version 6.2.2 using optimization level 2. Additional libraries are linked only as necessary for POSIX threads and software transactional memory.

[2]All included measurements were from executions run solo on a DAS4 node with 16 logical cores: 2 sockets, 4 cores per socket, 2 threads per core.

These decisions led to a particular implementation used in this section. Its distinctive properties are enumerated below:

1. N worker threads are spawned at initialization, where N is an environment variable[3]. Each worker persists over the entire execution and works on a subset of the matrix rows.

2. Each worker must decide whether or not the simulation meets the *convergence condition*. This condition is defined as $\forall x(\neg\Delta(x))$ where predicate Δ is 'has changed at least δ since the last iteration' and δ is some small numeric parameter. Before the barrier, workers will mark in a *shared vector* V whether cells within their rows satisfy the convergence condition (as a boolean). After the barrier, all workers traverse V to decide whether or not the condition is satisfied globally. Thus, all threads reach consensus using one barrier per iteration.

3. Threads can be on either side of the 'work' within some arbitrary iteration N, but the iteration barrier prevents them from diverging further. Still, this results in two arbitrary threads T_1 and T_2 potentially being on either side of the 'work', with T_1 writing to V (after the work) before T_2 has had a chance to read it (before the work). This is avoided by V being represented by two vectors, V_1 and V_2, each used as 'V' in an alternating fashion by swapping pointers, thus avoiding a data-race.

2.2 Hypotheses

This subsection outlines our expectations and hypothesis expected of our experimentation findings to come.

Our choices about the high-level algorithmic approach to the temperature dissipation problem are largely informed by the findings of our previous two reports, particularly the findings from OpenMP experiments. Many properties of the OpenMP implementation are carried over into this one (for example, statically partitioning the rows of the surface matrix over threads), as the problem and the tools we have to solve it are both very much the same. However, the code itself looks vastly different, owing to the increased granularity needed for manually managing threads.

[3]The use of an environment variable was beneficial for automated testing. For the end user, this number would default to a constant.

We expect that, as the application programmers, we will be able to leverage highly minimal and efficient solutions for these granular details, leading to a program more performant than that achieved by the compiler-built OpenMP implementation. However, the speed of our implementation is still fundamentally governed by the same properties as the OpenMP implementation; as such, the speedup is expected to be moderate. Our expectations are paraphrased in Hypothesis 1.

Hypothesis 1: *Runtime of the PThreads temperature dissipation implementation will be faster than that of the OpenMP implementation, but not significantly so.*

Of course, it is also possible that our implementation makes more foolish choices than those made by the compiler for the OpenMP version, it being able to leverage the knowledge and experience of many seasoned compiler programmers with more experience than ourselves. Nevertheless, we remain optimistic.

2.3 Observations

For the experiments in this section, inputs with dimensions in $\{100 \times 100, 1000 \times 1000, 4000 \times 4000\}$ to explore the effect in changing the scale of the image. We also explore images of constant scale (having 16 million cells) but with varying dimensions in $\{4000 \times 4000, 20000 \times 500, 500 \times 20000\}$.

To understand the performance of the PThreads implementation, its performance (in GFLOPs) was measured in comparison to that of the OpenMP implementation. In all cases, the input problem was linearly-scaled versions of the images shown in fig. 1. These images were chosen as they exemplify interesting edges and gradients that would result in a non-trivial heat simulation.

Henceforth, we refer to 'the OpenMP implementation' and 'the PThreads implementation' as O and P respectively for brevity.

For larger input images, consisting of 16 million pixels shown in figs. 2, 4 and 5, both O and P performed very similarly. These three cases represented the same problem size, but with work partitioned differently over worker threads (as the input matrices have different dimensions). There is little appreciable difference in observed performance between the three problems. In all three cases it can be seen that P outperforms O in the case of having 1 and 2 threads.

Counter-intuitively, fig. 4 shows P outperforming the sequential implementation when given only a single thread. It is unknown whether this finding is the result of an out-lier, or whether there is some unintentionally beneficial change to the logic for the P implementation from the sequential one that resulted in this situational (and relatively useless[4]) speedup.

Fig. 6 shows that for a small input, P is not much better than the sequential implementation, particularly given more threads. Worker threads for P use a multi-stage synchronization process when deciding whether or not to keep going. Creating the vectors for this process cost constant time for the run, per thread. O's approach to dealing with this problem likely does without these time investments and thus is able to scale better even for such a small problem. For both implementations, they performed poorly given 32 threads for this problem; with so little work to do, the thread-spawning overhead of the redundant workers overshadowed any benefits achieved from parallelism.

In most experiments experiments, there was not much speedup to be had with more than 8 threads, as demonstrated in fig. 3. This is due to the architecture of the DAS4 node, where the 16 logical cores contend with the ability to do only 8 floating point operations in parallel. For the 'moderately-sized' problem instance with dimensions 1000×1000 (seen in fig. 7) we see both implementations behaving somewhat similarly, but P dipping just at the wrong moment, leading to O acquiring the best result of nearly 10 GFLOPs. P peaking at 16 threads rather than 8 threads here would suggest that P does significant work other than FLOPs.

From the results seen in figs. 2 and 4 to 7, we see that while they are quite similar, there are cases where either O or P significantly outperformed the other. Thus we unfortunately **reject** Hypothesis 1. As the performance between P and O are comparable, it would seem that the manually-implemented P is similar to the implementation generated by the compiler's interpretation of the OpenMP pragmas. Whether this failure is a credit to GCC's OpenMP support or a sign that there are still performance optimization opportunities for P will require future work.

[4]We assume that there is not much demand for a multi-threaded program intended for sequential use.

(a) Initial temperature (b) Conductivity

Figure 1: Initial temperature and conductivity inputs for all experiments in Section 2.

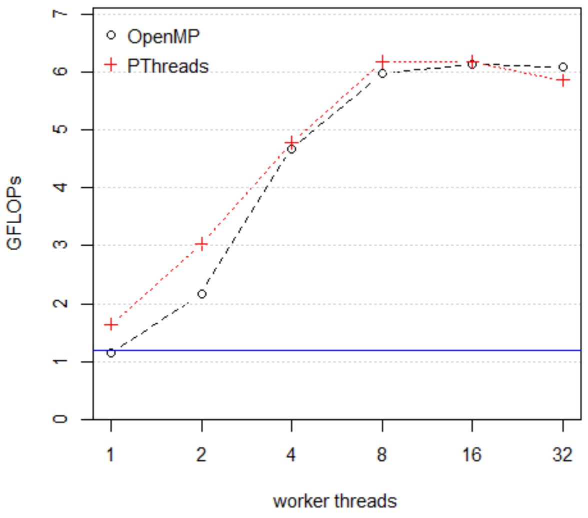

Figure 2: GFLOPs measured for OpenMP and PThreads implementations in response to the number of threads on an input program with dimensions 4000×4000. For comparison, the GFLOPs of the sequential implementation is shown as a horizontal blue line.

3 Pipe Sort

In this section we explore a program which exemplifies the (albeit contrived) use case of sorting elements using a nontrivial scheme for threads and their data-dependencies.

'Pipe sort' is a toy example of a sorting algorithm which makes use of a chain of logical 'units', each

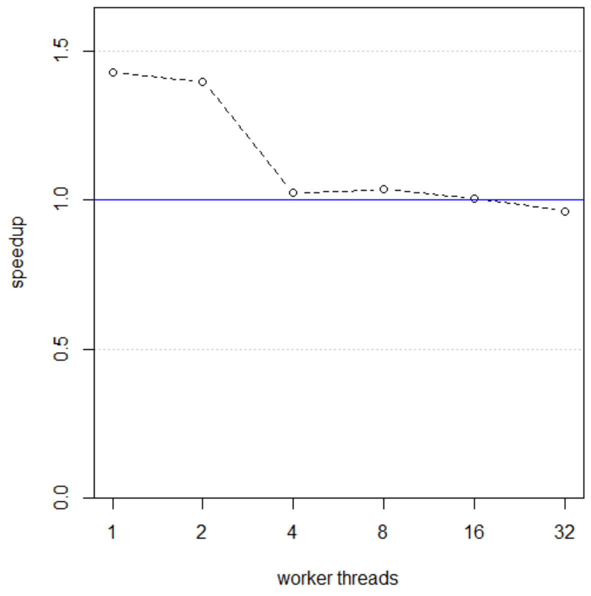

Figure 3: Speedup of P relative to O for an input image of dimension 4000×4000 (Seen in fig. 2) plotted over a varying number of threads.

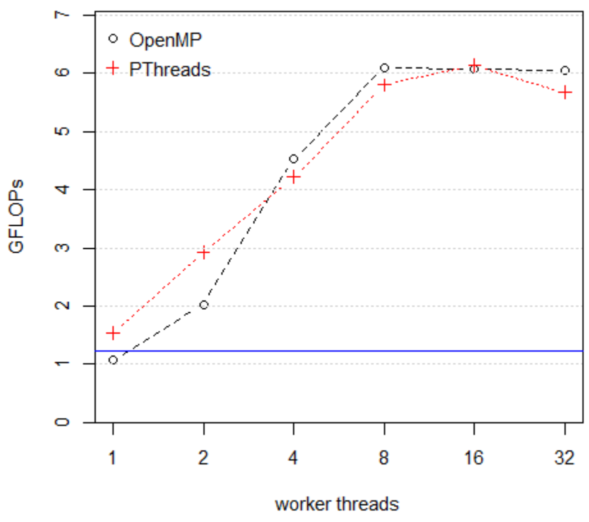

Figure 5: GFLOPs measured for OpenMP and PThreads implementations in response to the number of threads on an input program with dimensions 800×20000. For comparison, the GFLOPs of the sequential implementation is shown as a horizontal blue line.

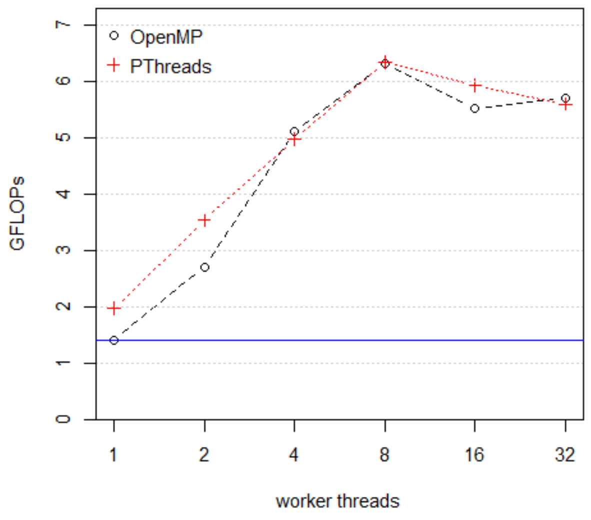

Figure 4: GFLOPs measured for OpenMP and PThreads implementations in response to the number of threads on an input program with dimensions 20000×800. For comparison, the GFLOPs of the sequential implementation is shown as a horizontal blue line.

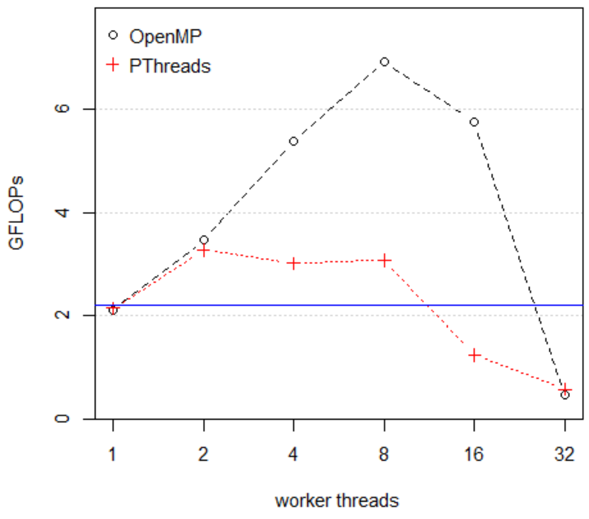

Figure 6: GFLOPs measured for OpenMP and PThreads implementations in response to the number of threads on an input program with dimensions 100×100. For comparison, the GFLOPs of the sequential implementation is shown as a horizontal blue line.

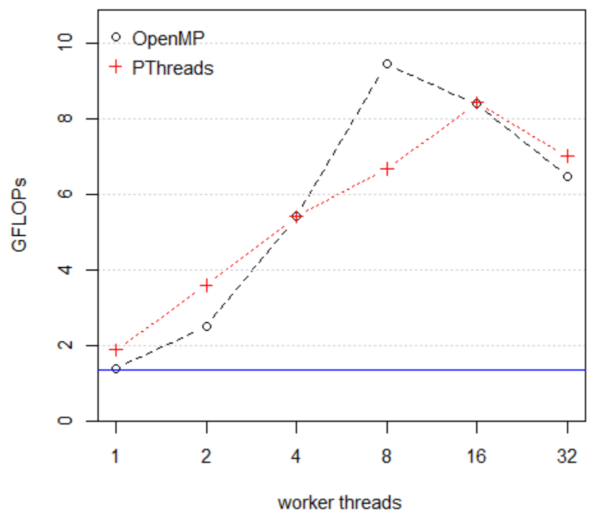

Figure 7: GFLOPs measured for OpenMP and PThreads implementations in response to the number of threads on an input program with dimensions 1000×1000. For comparison, the GFLOPs of the sequential implementation is shown as a horizontal blue line.

of which corresponds to an independent worker thread. These units exhibit a continuous 'pipe' of producer-consumer couplings with their neighbors, each continuously (and independently) consuming values from their predecessor and producing values for their successor. From relatively simple code, where each logical unit only takes care of withholding *the smallest element* they have seen so far while relaying the rest, the complex behavior of *sorting* emerges.

Our implementation adheres to the specification (as paraphrased by the paragraph above). Within these constraints, still many design decisions are possible. What follows is a more specific description of our implementation.

3.1 Implementation

Here we outline specifics of our implementation, as they are relevant to understanding the performance measurements in the section to follow.

1. We define a 'PipeBuffer', a circular, bounded[5] buffer for integers. The 'owner' of the buffer *produces* for their successor by populating their *own* buffer with elements. This successor *consumes* these elements. The PipeBuffer structure contains two semaphores which track the occupied/available space in the buffer to preserve correctness.

2. The generator thread spawns a 'comparator'. Next, it generates and produces N random[6] values where N is a runtime argument. Next, the generator produces two END symbols[7]. Finally the generator thread blocks until it receives the end-of-program signal.

3. Comparator threads spawn given a pointer to the PipeBuffer of their predecessor (used for consuming). They also create their own PipeBuffer (for producing).

4. Comparator threads produce all values they consume except for M, the smallest value they encounter. If the comparator encounters an END before any other value, they specialize themselves as the output thread (see below). Otherwise, comparators forward all subsequently-consumed values. Once they encounter the second END, they forward M and then join() their successor thread.

5. The output thread has access to a predecessor PipeBuffer. It greedily prints all values it consumes. Once the output thread encounters an END, it sends the end-of-program signal and terminates.

Most noteworthy is the choice of an end-of-program signal between the output thread and the generator thread. A simpler approach has all threads (including the generator thread) calling join() on the thread-handle of their successor once their work is completed. This can unfortunately lead to a needless wait between the output finishing its work and the (main) generator thread finally being able to join, as the entire linear chain of comparators must join in sequence first.

[5]The size of PipeBuffers is a compile-time constant.

[6]The END symbol is represented as a specific integer constant (zero). Thus, the same integer is prohibited from being generated by the random number generator. As the specification does not explicitly define the domain of random integers, this minimal and extensible approach was favored.

[7]These symbols are produced just like other generated values, but carry extra semantic meaning which is made clear later.

Intuitively, one would like the generator to be able to join() the output thread's handle directly, but the output thread is not yet created until some distant point in time (potentially long after the generator thread finishes working). To approximate a direct join, a single global semaphore is initialized with a capacity of zero. The generator decrements it upon finishing, and the output thread increments it upon finishing. In effect, this allows the generator (and the program) to wrap up safe in the knowledge that the output has been completed and flushed *without* many intermediary comparator threads finishing the arduous process of joining each other sequentially.

As a consequence, the generator is a special case of requiring a PipeBuffer that might outlive it. Thus, the generator's PipeBuffer is passed by reference, and owned by the main function (unlike comparator threads who store their own PipeBuffer directly within their stack frame). This is necessary, as upon receiving the end-of-program signal, the generator's successor might still be running, and attempting to consume from the generator's PipeBuffer. If this crucial buffer is stored on the stack within the frame of the generator's function call, this read might occur within invalid memory[8].

This implementation raises a question: What is the ideal size of the PipeBuffer for each worker? Larger buffers clearly require more memory, but they allow the producer to out-pace the consumer if necessary (reducing the frequency of producers blocking, waiting for consumers to drain the buffer). As a preliminary experiment, the change in the system's runtime was observed as a function of the chosen size of this input. For the experiments in this section, the random number generator is deterministically seeded an arbitrary constant. Note that while this does predetermine the movement patterns of generated values through the pipeline, the inter-leavings of operations across threads remain out of our control.

3.2 Hypotheses

Analysis of the algorithm leads to the postulation of Hypothesis 2.

[8]As no further function calls occur from main(), this is unlikely to be an issue, but is rather a hairy detail that may potentially cause problems in unusual circumstances. The workaround is inexpensive, and thus we include it to be safe.

Hypothesis 2: *Runtime of pipe sort will be quadratic with respect to the input sequence length.*

Each input element results in a new worker thread, each having to produce, compare and consume elements in order $\mathcal{O}(n)$, where n is the length of the input sequence. The limited, but non-trivial number of threads that can be allocated to CPUs is expected to muddy this quadratic result somewhat[9], but we expect that 16 logical cores is significantly small in comparison to the length of the input sequence; this quadratic relationship will still be discernible.

It is unclear to what extent longer PipeBuffers will help to decrease the program's runtime. This is due to the speed of execution being largely dependent on the *order* threads will be allocated CPU time quantums. Consider, for instance, the worst-case scenario where the entire pipeline cannot finish working all because one comparator thread starves. Ergo: Hypothesis 3.

Hypothesis 3: *Runs using longer PipeBuffers will reduce runtimes drastically for shorter buffers, then with decreasing effect with larger buffers.*

We make no claims other than that longer PipeBuffers are consistently beneficial until they aren't. At some stage the buffers are unlikely to be filled, and this speedup will taper off.

3.3 Observations

Five values for the length of the PipeBuffer were selected, given by $x \in \{1, 4, 16, 32, 64\}$, whose measurements are given corresponding aliases of the form 'bplen=x'. fig.s 8 and 11 show the measured runtimes in response to varying the length of the input sequence. Eyeballing the curve seen in fig. 8 would suggest that the runtime pblen=1 is approximately in order $\mathcal{O}(n^2)$ in response to the input sequence length. This intuition is supported by fig. 9, where it can be seen that two arbitrary functions in $\mathcal{O}(n^2)$ and $\mathcal{O}(n^{2.2})$ model this runtime accurately. As such, we **accept** Hypothesis 2.

[9]Time complexity in $\mathcal{O}(n^2)$ is clear with a sequential implementation. With infinite CPUs, this parallel algorithm would approach something more akin to $\mathcal{O}(n)$. However, with very many more input elements than cores, we are closer to the sequential situation than the parallel one.

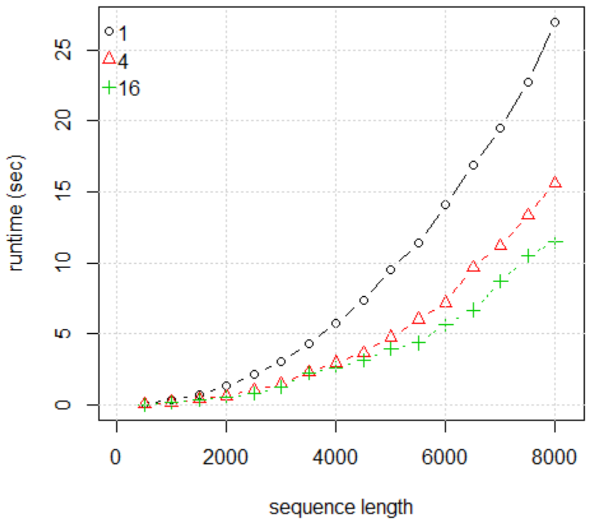

Figure 8: Runtimes for pipe sort implementations with PipeBuffer lengths in $\{1, 4, 16\}$ in response to input sequences with given length.

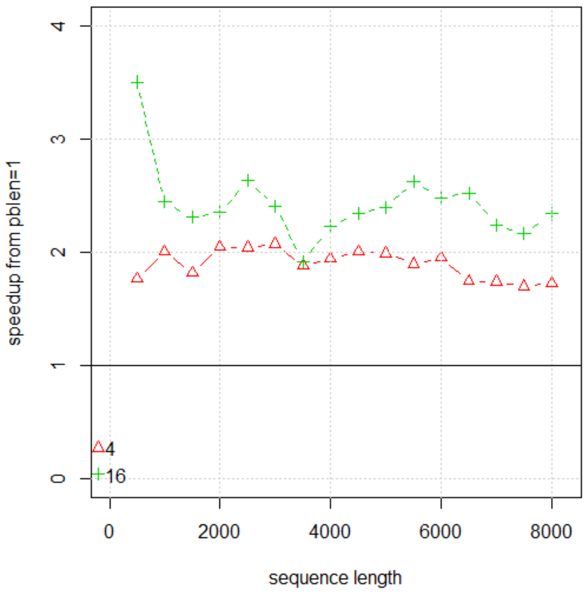

Figure 10: Speedup of pipe sort implementations with PipeBuffer lengths in $\{4, 16\}$ relative to PipeBuffer length 1 (as shown in fig. 8) plotted over various input sequence lengths.

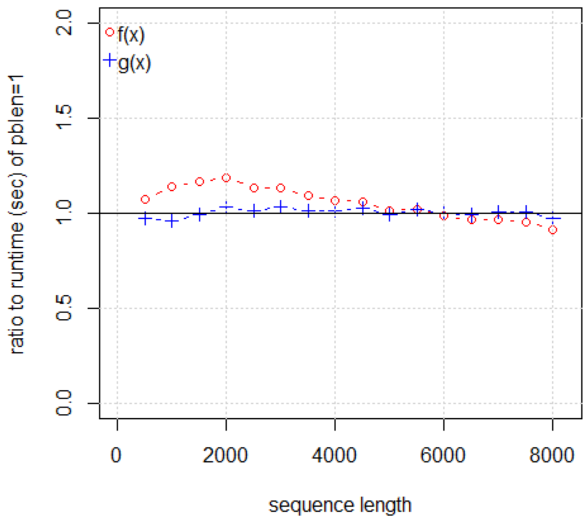

Figure 9: Functions $f(x) = \frac{x^2}{2.6\times10^5}$ and $g(x) = \frac{x^{2.2}}{1.5\times10^7} + \frac{x}{1.7\times10^4}$ normalized to runtime of pblen1 implementation for pipe sort (ie. the ratio to pblen=1), plotted over various input sequence lengths.

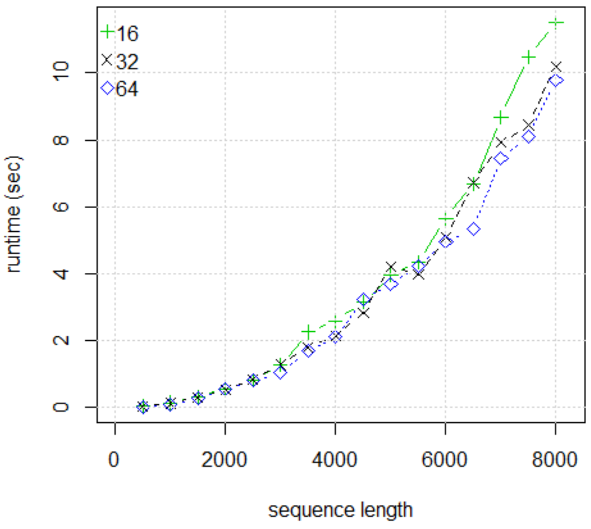

Figure 11: Runtimes for pipe sort implementations with PipeBuffer lengths in $\{16, 32, 64\}$ in response to input sequences with given length.

Fig. 8 shows that stepping up from the smallest to moderately-sized PipeBuffers consistently results in speedup, which is more apparent when viewing the same data in the form given by fig. 10. With additional increase to the buffer size (shown in 11) there are diminishing returns to the increased speedup. This is intuitive, as an increased buffer size does not do anything to increase the throughput of a comparator if it empty anyway (which occurs whenever the *producer* of the PipeBuffer is slow). Assuming arbitrary allocations of threads to CPU time quantums by the operating system, the producers hold up progress in this way about *half* the time[10], regardless of the buffer size.

Fig. 11 shows that pblen=64 is unique in being the only variation that did not outperform its shorter-length counterparts consistently (More often than not having a larger runtime than pblen=32). As the differences are not significantly large, nor entirely consistent, this finding might suggest nothing more than the fact that there is *no significant advantage* to increasing buffer size beyond approximately PipeBuffer length of 32. From these findings, we also **accept** Hypothesis 3, but fig. 11 suggests that more experiments may be required to confirm that plen=64 (and beyond?) are not *significantly* worse after all.

To test the limits of the pipe sort program, the superlative pblen=32 version was used to sort the largest input sequences. It was able to sort 10000 elements in 17.88 seconds, 12000 elements in 33.17 seconds, 14000 elements in 49.69 seconds, and 16000 elements in 60.80 seconds.

4 Grayscale Histogram

This section explores a relatively simple program: Given an input image of grayscale pixels (with values in range 0–255), compute the histogram of pixel values.

Within this simple framework, this section explores different implementations of the same program, each using a different mechanism to facilitate safe, concurrent modification of the histogram object. This resulted in five implementations, outlined in Table 1 along with short labels for brevity in this section.

[10]Amusingly, this tapering speedup can be seen as an unconventional manifestation of *Amdahl's law*, as a constant component of the runtime inhibits the benefits of downscaling the rest of the runtime.

4.1 Implementations

As it is left unspecified, we approach the problem in the data-parallel approach: Each worker thread inspects a subset of the image's rows and contributes its findings to a single histogram. This approach was chosen for the same reason as it was in Section 2. The versions of the program shown in Table 1.

Table 1: Grayscale histogram implementation variants compared in Section 4. Each variant is provided a label for identification as well as the description of its definitive concurrency mechanism.

Label	Description
M	Threads write to the global histogram after acquiring a mutex lock protecting the relevant pixel value.
S	Threads write to the global histogram after decrementing a semaphore initialized with capacity 1 associated with the relevant pixel value. Afterward, the semaphore is incremented again.
A	Increment operations on the shared histogram's values are atomic.
T	Increment operations on the shared histogram's values are in transactional memory blocks.
L	Each thread builds its own local histogram. Once finished, threads acquire a global lock and merge their histogram into the global histogram.

The subsections to follow explore the performance of this system in response to various properties.

4.2 Hypotheses

A priori, we put forward the following hypotheses for the experiments in this section. They are intended not only to suggest our expectations for the results of the experiments to come, but also to guide the exploration of the search space in the direction of answering interesting questions.

Hypothesis 4: *M and S will always achieve similar results.*

Hypothesis 5: *L will be fastest under all conditions.*

Hypothesis 6: *Images having large quantities of possible pixel values will affect all implementations except for L.*

Hypothesis 7: *A will outperform M and S but not L.*

Hypothesis 8: *L will have a higher optimal choice for number of threads than the others.*

Although they require different logic for the programmer to use and have a slightly different API, we believe semaphores will incur similar costs to mutex locks. Ergo, Hypothesis 4.

While all implementations deal with contention on the shared resource (the histogram structure), only L takes an approach in the same vein as the *dynamic programming paradigm*: trading in a bit of memory for a lot of speed. L has very modest need for concurrency primitives compared to the others implementations. This contention is not only expected to be smaller (supporting Hypothesis 5), but it is expected to be a smaller *proportion* of the overall runtime too and bottlenecking the performance less (leading to Hypothesis 8). With a unique independence from the granularity of the pixel-wise locks, L is expected to handle the case of few unique pixels the best (Hypothesis 6).

The cost of atomic operations in comparison to locks is unknown to us *a priori*. However, this problem's critical section maps perfectly to the use case of atomic variables; we expect that A would perform more favorably than its rivals, leading to Hypothesis 7.

4.3 Observations

This program exposes a very large experimentation space, as there are several implementation variants to compare, many possible relevant properties of input images as well as some in-program parameters. A full parameter sweep is infeasible. As such, we attempt to explore the performance response to parameters in stages, each time attempting to isolate one variable and affixing that variable at an appropriate value for future stages.

4.3.1 Image Type

As a starting point, it was interesting to see if the runtime of the implementations depended very much on the *structure* of the pixel data. A collection of constructed image types was used for the experiment, enumerated below. In these descriptions, C defines a numeric input parameter for the experiment. In effect, C determines the number of possible unique values present in the input image AKA the 'value range'.

1. **random**
   Pixel values are independently sampled from a uniform distribution[11] on the interval $[0, C]$.
2. **h_gradient**
   Pixels values form a linear gradient horizontally. The leftmost column's cells have value 0, and rightmost column's cells have value C.
3. **v_gradient**
   Pixels values form a linear gradient vertically. The top row's cells have value 0, and bottom row's cells have value C.
4. **striped**
   Pixel values are given by $i \bmod C$, where i is their column index, starting at 0. This results in vertical stripes, or a 'sawtooth' linear gradient horizontally.

The experiments were run with the full color range ($C = 255$), with input images with dimensions 10000×10000 and with 8 threads.

As seen in figs. 12 and 13, it is most immediately obvious that T is significantly slower than the other implementations, with an apparent constant extra runtime. We also observe that L is faster with fewer threads for this problem. This is likely due to the work being overshadowed by the thread-spawning overhead.

All implementations ran the slowest for 'random' images, with 'h_grad' coming second. This makes sense, as 'v_grad' works well with the row-wise splitting of threads' work, resulting in most threads working on a specific sub-range of $[1, C]$. The 'striped' images also resulted in low runtimes;

[11] The canonical use of something like 'rand() % n' to generate a random number on the interval $[0,n)$ in C does not truly result in uniformly-distributed samples (without bias). However it is close enough for our purposes.

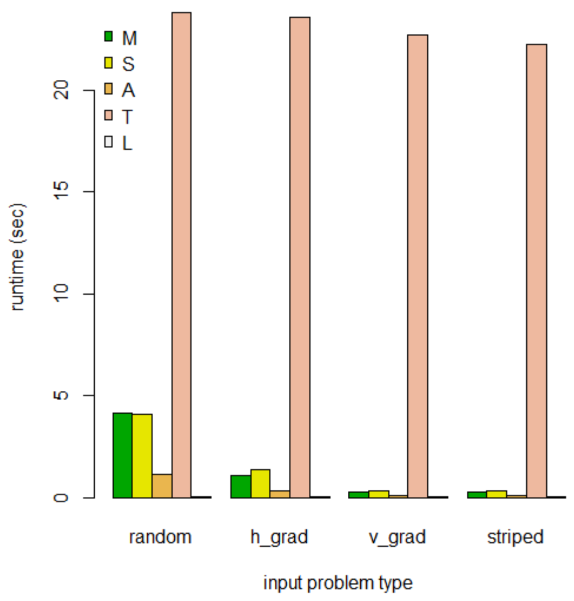

Figure 12: Runtimes using 8 threads and $C = 255$ on an input image with dimensions 10000×10000. Bar groups discern between different image 'types', with each group containing measurements for implementations $\{M, S, A, T, L\}$.

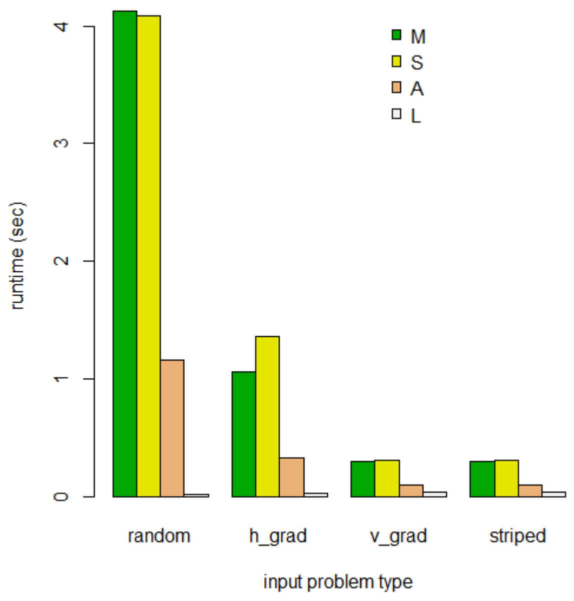

Figure 13: Fig. 12 plotted without T to make the other bars more visible.

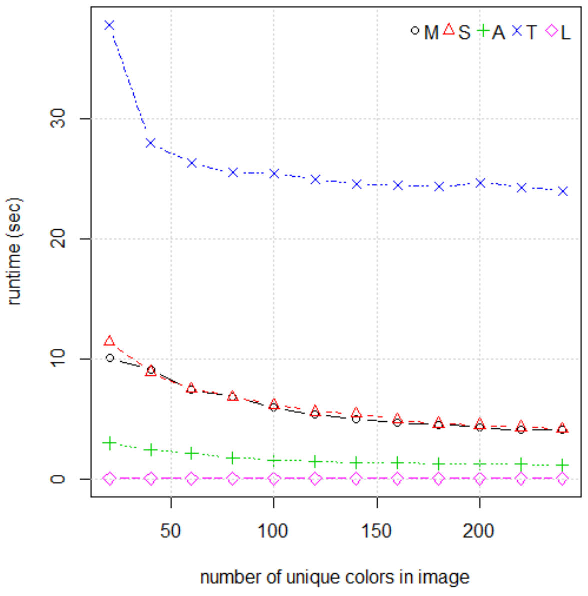

Figure 14: Runtimes for implementations $\{M, S, A, T, L\}$ on 'random' type input images with dimensions 10000×10000 using 8 threads, plotted over the number of unique image pixel values (C).

this would occur if threads would move through the columns of their work in a staggered fashion. Any two threads would experience no contention if they are separated by only one *column*. If this isn't the case, one thread would momentarily block the other and go ahead by a column, as such it can be said that this 'staggered' thread progression is *stable*. As hypothesized, A and S are comparable, with A faster and L faster still for all runs.

As it is the most 'fair' measurement, and it being the least predictable, we use 'random' images for future experiments.

4.3.2 Image Color Range

In this section we are interested in understanding the effect on altering parameter C has on runtime, where C defines the number of unique pixel values in the input image. With small values for C, the same amount of work is being done using fewer pixel 'buckets'. As such, lower C would result in higher contention.

Fig. 14 shows the results of the experiments with 'random' images with dimensions 10000×10000 and using 8 threads.

Runtimes of L stay around a modest 17 milliseconds for all runs, with a standard deviation of less than half a millisecond. This consistency makes sense, as it is the only implementation whose synchronization mechanism's granularity is not tied to C. The ordering of $[L, A, \{M, S\}, T]$ in ascending runtime consistently across runs is unchanged from the previous experiment. As expected, all implementations other than L benefit from the decreased contention of a higher choice of C. This results in diminishing returns at large values of C as the relationship on contention is *geometric* and not *linear* with respect to C.

As the results of this section are not surprising, we affix C at the full (standard) image pixel range of 0–255 henceforth.

4.3.3 Image Scale and Threads

With C and the image 'type' affixed at 255 and 'random' respectively, this section explores how the various implementations behave in response to the *scale* of the input image. As images until now were large, and this section allowed experimentation on images that were small-to-large, this section also doubles down[12] on identifying a relationship with the number of threads used.

As seen in figs. 15 to 19, L's runtime continues to dwarf that of its counterparts. For all implementations, there is also no added benefit in threads in excess of 16, with all runtimes going up. The extreme climb of L here suggests that the cost of simply *having* more threads causes immense slowdown, rather than just these threads experiencing increased contention.

To better understand what is happening to the faster implementations, fig. 20 shows the same data as fig. 19, but with *both* axes plotted logarithmically. With this view, one can observe[13] that M and S behave incredibly similarly in all instances, with A being consistently faster. Even at this scale, L achieves rock-bottom runtimes, hundreds of times faster than its counterparts.

[12]Both variables are explored independently by performing a 2D parameter sweep on combinations of choices for image scale and number of threads.

[13]The logarithmic plots of the other images from this subsection produced no new information, and were thus omitted. The appendix contains the raw data for inspection.

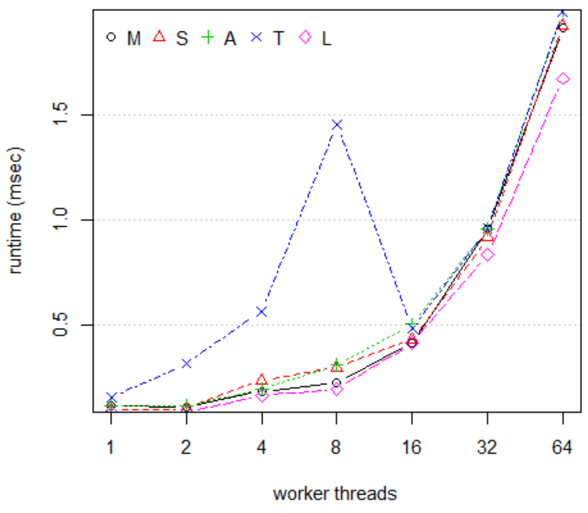

Figure 15: Runtimes for implementations $\{M, S, A, T, L\}$ for a 'random' type image with $C = 255$ with dimensions 10×10, plotted over a (logarithmically) increasing number of worker threads.

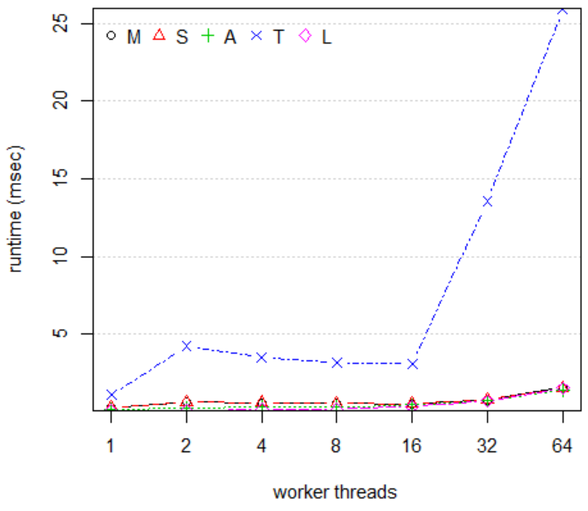

Figure 16: Runtimes for implementations $\{M, S, A, T, L\}$ for a 'random' type image with $C = 255$ with dimensions 100×100, plotted over a (logarithmically) increasing number of worker threads.

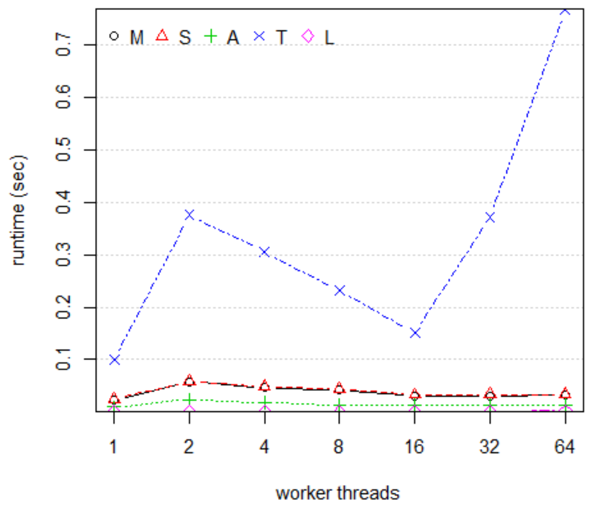

Figure 17: Runtimes for implementations $\{M, S, A, T, L\}$ for a 'random' type image with $C = 255$ with dimensions 1000×1000, plotted over a (logarithmically) increasing number of worker threads.

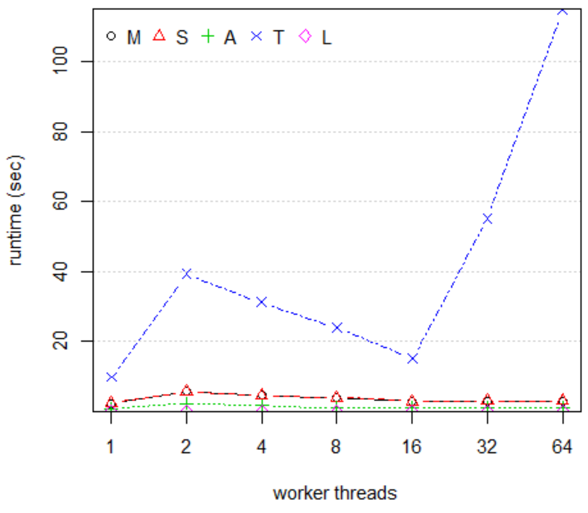

Figure 19: Runtimes for implementations $\{M, S, A, T, L\}$ for a 'random' type image with $C = 255$ with dimensions 10000×10000, plotted over a (logarithmically) increasing number of worker threads.

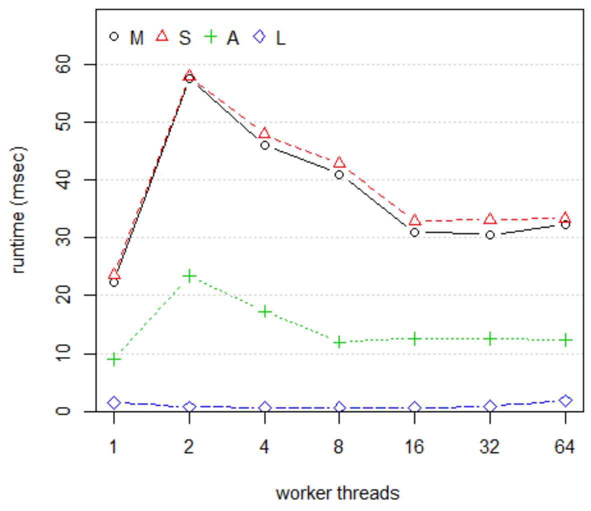

Figure 18: Fig. 17 without T to make the smaller values more visible.

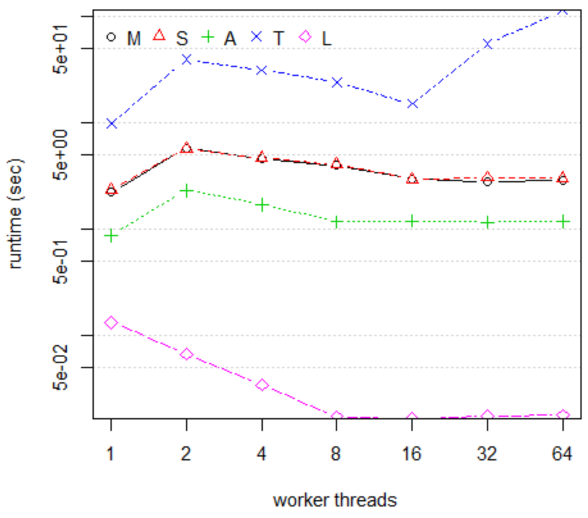

Figure 20: Runtimes in Fig. 19 re-plotted with *both* axes scaled logarithmically.

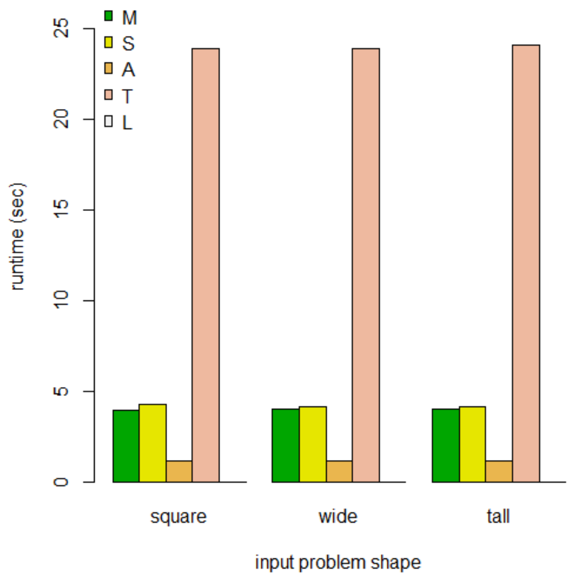

Figure 21: Runtimes for 'random' input problems with $C = 255$, 8 threads, dimensions of 10000×10000 for different shapes of images discerning between bar groups, with implementations $\{M, S, A, T, L\}$ measured per group.

The results of 15 are the only ones that don't match the others, with all implementations lining up neatly for large counts of threads. This may be the result of it being a *very* small input image (only 100 pixels total!) which results in lucky cases of no collisions.

With the exception of the smallest image, we see that the behavior of all implementations is consistent over image sizes. Also, all implementations were behaving as expected for thread counts. Therefore, for the final experiment we maintain the status quo of using an image with dimensions 10000×10000 and 8 threads.

4.3.4 Image Shape

Finally, we attempt to identify whether the *shape* of an input image plays a role in its runtime (where shape is meant as the ratio between rows and columns). We concentrate on three image shapes: 10000×10000 ('square'), 200000×500 ('wide'), and 500×200000 ('tall'). All these images have the same number (100 million) pixels, but different shapes.

As can be seen in fig. 21, there are no surprises waiting in these results. At these scales, the number of rows vastly outnumbers the number of threads, and there is no apparent impact of iterating over more, shorter rows than fewer, longer ones. As such, the results of all three experiments exemplify the patterns found thus-far, with A and M behaving similarly, A performing better, L performing best and T being blown out of the water by the others by an order of magnitude.

With no further experiments to conduct, we reflect on our hypotheses:

- **Hypothesis 4: Accept.**
In all cases, the measurements for M and S were very close, with most of the cases having them nearly indistinguishable, particularly with larger input problems. This result was expected, as they are both lock-based concurrency mechanisms that rely on the same principles.

- **Hypothesis 5: Accept.**
L was fastest, beyond a shadow of a doubt. Although the fine-grained locking used by the other implementations provided speedup with more threads, L's solution of avoiding locking as much as possible made the largest difference.

- **Hypothesis 6: Accept.**
Changing the C parameter did not affect L at all, but did have a significant impact on the other implementations. Proportionately, T experienced the second-smallest impact from changing C, but that is mostly due to T just being terribly slow all the time.

- **Hypothesis 7: Accept.**
The degree to which A outperformed M and S speaks volumes for the nature of this problem, as most of the implementations were all attempting to *simulate* atomic operations anyway.

- **Hypothesis 8: Reject.**
All threads experienced their optima when using all 16 threads. It was expected that as the number of unique colors (and thus, 'boxes') became comparable to the number of threads, threads would begin fighting over

boxes and L's avoidance of relying on this approach would give it the edge. At this scale we did not observe L distinguishing itself in this manner. In order to check this definitively, there would have to be a follow-up experiment with a very very large number of threads, but with various very small choices for C.

The most striking observation overall is how terribly inferior T was compared to the other method. Software transactional memory is known as being best used in cases of low contention, and the take away from this section should be that the contention replicated here is clearly not low enough. In addition, it is clear that the GCC support for software transactional memory has a long way to go. A was well-suited to the task, as it most neatly mapped its solution to the problem, and its performance reflected that. At the end of the day, the excellence of L just goes to show that no amount of clever performance tricks can outshine the effect of choosing a smart algorithm to begin with.

5 Conclusion

PThreads allows the programmer a granular level of control and clarity that allows for implementation of programs (such as pipe sort) that would otherwise be impossible to express as-intended in a higher-level manner (such as OpenMP). In some cases, even where a program could be implemented in OpenMP, the explicit nature of PThreads makes it harder to overlook performance critical design decisions.

However, many problems exist for which the implementation is best done leveraging a well-understood paradigm such as coarse-grained data parallelism. For these problems, the control provided by PThreads isn't likely to provide much speedup over a well-programmed OpenMP implementation, as the programmer would be hard pressed to beat the compiler at doing something it was intended to do. In addition, the PThreads implementation is likely to be more obtuse, verbose and difficult to maintain.

In the end, regardless of the efforts of the programmer to 'sand the corners off' an existing program, nothing can beat choosing the right algorithm to begin with.

Appendix

Table 2: GFLOPs measured for runs of the sequential heat dissipation program of Section 2. Columns isolate input problem instances.

	100x100	1000 x 1000	4000 x 4000	20000 x 800	800 x 20000
1	2.207373	1.341317	1.201316	1.406545	1.678821

Table 3: GFLOPs measured for runs of the OpenMP heat dissipation program of Section 2. Columns isolate input problem instances. Rows discern between the number of worker threads used.

	100x100	1000 x 1000	4000 x 4000	20000 x 800	800 x 20000
1	2.112367	1.315744	1.151074	1.404204	1.069346
2	3.521687	2.49465	2.175861	2.701771	2.041937
4	5.39931	5.36848	4.688237	5.059101	4.539618
6	6.080945	7.863781	6.015544	6.218255	5.676172
8	6.933983	9.319562	6.294753	6.306051	6.330631
16	5.789277	8.545961	6.136148	5.512627	6.060978
32	0.4242867	6.424931	6.063079	5.680519	6.068174

Table 4: GFLOPs measured for runs of the PThreads heat dissipation program of Section 2. Columns isolate input problem instances. Rows discern between the number of worker threads used.

	100x100	1000 x 1000	4000 x 4000	20000 x 800	800 x 20000
1	2.142672	1.876752	1.645253	1.971684	1.538667
2	3.260617	3.584637	3.122784	3.528706	2.926755
4	3.027172	5.44944	4.775084	4.971203	4.329945
6	3.76366	6.998688	6.082269	6.281458	5.558572
8	3.023205	9.186889	6.330166	6.241809	5.946538
16	1.146917	8.593669	6.127539	5.680487	6.110788
32	0.6456335	6.966096	5.854185	5.585639	5.689754

Table 5: Runtimes (in sec) measured for pipe sort in Section 3. Columns discern between different choices of the length of the PipeBuffer. Rows discern between lengths of the input sequence.

	1	4	16	32	64
500	0.090749	0.051885	0.026063	0.021717	0.021245
1000	0.307930	0.152869	0.130719	0.123440	0.080729
1500	0.721242	0.422515	0.397847	0.233422	0.291699
2000	1.250578	0.663801	0.496830	0.602443	0.502782
2500	2.058815	1.047539	0.977626	0.852828	0.670080
3000	3.082699	1.549832	0.849148	1.615687	0.929234
3500	4.209787	2.640351	2.765728	1.904900	2.066642
4000	5.892613	3.350248	1.974003	2.516606	2.295241
4500	7.418398	3.494844	2.625834	2.287460	3.029472
5000	8.793933	4.738094	3.697062	3.240306	4.496875
5500	11.193493	6.336459	3.815153	3.279570	4.304335
6000	14.474953	6.499371	5.484541	4.622364	5.022379
6500	16.491075	9.219182	6.397772	5.211816	4.752167
7000	19.022253	10.311392	8.702629	10.468972	6.779717
7500	22.413440	13.162874	9.975965	8.520490	9.113678
8000	26.203247	16.174314	10.289865	9.054436	9.349862

Table 6: Runtimes (in sec) with each implementation from Section 4 in its own column. Rows discern between image ‘type’

	M	S	A	T	L
random	4.123781	4.091154	1.164217	23.800371	0.017255
h_grad	1.061668	1.362748	0.332304	23.572875	0.023155
v_grad	0.294960	0.305650	0.100058	22.724759	0.033820
striped	0.294862	0.305601	0.100046	22.280204	0.034049

Table 7: Runtimes (in sec) with each implementation from Section 4 in its own column. Rows discern between the number of unique pixel values in the input image.

	M	S	A	T	L
20	10.022897	11.370493	2.962965	37.72076	0.0173568
40	9.097914	8.897203	2.393572	27.94372	0.0174582
60	7.409494	7.472608	2.098614	26.27936	0.0175106
80	6.815896	6.79417	1.726932	25.46338	0.0172924
100	5.957639	6.165873	1.580966	25.41662	0.0173182
120	5.325137	5.575528	1.451363	24.89226	0.0185632
140	4.950547	5.408944	1.352638	24.53504	0.0173012
160	4.637107	4.897773	1.313444	24.42057	0.0171976
180	4.482347	4.580808	1.262312	24.30693	0.0173274
200	4.279206	4.469404	1.225209	24.61277	0.0172308
220	4.045274	4.293797	1.198787	24.25403	0.017314
240	4.123708	4.136281	1.163026	23.95505	0.0172114

Table 8: Runtimes (in sec) on an input ‘random’ image with $C = 255$ and dimensions 10×10 with each implementation from Section 4 in its own column. Rows discern between number of threads.

	M	S	A	T	L
1	0.0001178	9.66E-05	0.000117	0.0001576	8.88E-05
2	0.0001118	0.000102	0.000119	0.0003194	8.94E-05
4	0.0001882	0.0002396	0.000196	0.000567	0.0001668
8	0.0002282	0.0003012	0.0003118	0.001453	0.0001966
16	0.000415	0.0004342	0.0005046	0.0004856	0.0004154
32	0.000955	0.0009164	0.0009584	0.0009592	0.0008354
64	0.001908	0.0019228	0.0019938	0.0019866	0.00167

Table 9: Runtimes (in sec) on an input ‘random’ image with $C = 255$ and dimensions 100×100 with each implementation from Section 4 in its own column. Rows discern between number of threads.

	M	S	A	T	L
1	0.000305	0.0003152	0.0001648	0.0011192	0.0001016
2	0.0006364	0.0006684	0.0002674	0.004234	0.0001062
4	0.0006096	0.000613	0.0003422	0.0035188	0.0001754
8	0.0006118	0.0005606	0.0003612	0.0031674	0.0002036
16	0.0005346	0.0005416	0.0004808	0.003126	0.0003742
32	0.0007786	0.0007836	0.0007264	0.0135406	0.0007146
64	0.0015828	0.0015168	0.0014334	0.0259376	0.0015362

Table 10: Runtimes (in sec) on an input ‘random’ image with $C = 255$ and dimensions 1000×1000 with each implementation from Section 4 in its own column. Rows discern between number of threads.

	M	S	A	T	L
1	0.022405	0.0236058	0.0090336	0.0993042	0.0015428
2	0.057571	0.0578074	0.023437	0.3760052	0.0007978
4	0.046054	0.0478648	0.0173182	0.305596	0.000601
8	0.041005	0.0427436	0.0120222	0.2313696	0.0007192
16	0.0309272	0.0327796	0.01262	0.1518888	0.0006124
32	0.0305882	0.0331602	0.0126456	0.3717388	0.0008488
64	0.0322118	0.0333028	0.0123568	0.7689944	0.001894

Table 11: Runtimes (in sec) on an input ‘random’ image with $C = 255$ and dimensions 10000×10000 with each implementation from Section 4 in its own column. Rows discern between number of threads.

	M	S	A	T	L
1	2.223815	5.803117	4.585767	3.970317	2.958638
2	2.341383	5.737973	4.631882	4.043686	2.931873
4	0.8774524	2.3131688	1.6975672	1.1682630	1.1946538
8	9.896998	39.283848	31.221863	23.873245	15.099419
16	0.1327608	0.0670624	0.0343364	0.0172916	0.0165882
32	0.0305882	0.0331602	0.0126456	0.3717388	0.0008488
64	0.0322118	0.0333028	0.0123568	0.7689944	0.001894

Table 12: Runtimes (in sec) on an input 'random' image with $C = 255$ and 100 million pixels with each implementation from Section 4 in its own column. Rows discern between the 'shape' of the input image.

	M	S	A	T	L
square	3.9810506	4.2509868	1.1696016	23.8724362	0.0172594
wide	4.0383814	4.1660282	1.1628340	23.9135322	0.0171168
tall	4.0086656	4.1691364	1.1621104	24.0817204	0.0172078