Vrije Universiteit Amsterdam

Parallel Programming Practical
Chapel: Common Image Source Identification

Christopher Esterhuyse
2553295 – cee600
September 12, 2019

1 Introduction & Background

All individual digital cameras possess a persistent pattern in the non-uniformity of their light sensors as a byproduct of the manufacturing process. This property is called the 'photo-response non-uniformity' (PRNU). Given the ability to measure them directly, these patterns can be used to distinguish cameras as fingerprints can be used to distinguish humans. This non-uniformity also influences the values captured and ultimately reflected in the photographs the cameras take. Common Image Source Identification (CISI) is the process of teasing these patterns out of photographs and using them to reason about the *probability* that the photographs originate from the same camera. More precisely, the CISI problem accepts a sequence of N photographs and produces a correlation matrix of dimensions $N \times N$, with value (i, j) expressing the degree of correlation between images i and j.

Chapel is a modern partitioned-global-address-space (PGAS) language, developed by Cray. Boasting highly-integrated, concise and expressive concurrency and parallelism constructs, the language cements itself in the realm of super- and cluster computer programming with a focus on productivity. While the language is still under development, its powerful constructs can be used to great effect when building a highly-parallel program.

In this report, we detail the development of a parallel and distributed CISI solver in the Chapel language. We explore the design space for our problem in §2, and how properties both obvious and subtle guided the design of the final version. In §3, experimentation on various versions of the solver seek to tease out its performance properties. Finally, §4 provides the conclusion.

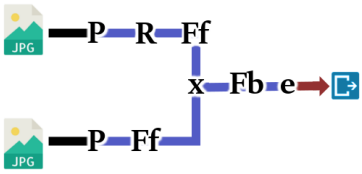


Figure 1: Illustration of the processes applied on two jpg images to produce the corresponding value in the correlation matrix, represented as a data-flow. Section 2.1 explains the nature of each process (marked with a unique symbol). The colors represent the data type in transit. Prior to P, jpgs are represented as RBG matrices. After e, a single floating-point number is output. In-between, data is represented as 2-dimensional complex matrices.

2 Design & Implementation

This section explores the observations about the CISI problem and Chapel language that lead to the final design of the optimized solver.

2.1 Solver Fundamentals

This section describes the space for all possible CISI solvers, implemented in Chapel.

2.1.1 Computation Pipeline

Before considerations of performance come into play, a CISI solver must correctly produce correlation values for all input images, assumed to be in jpg form. Computing the correlation for some image pair is a process independent of all other correlations. Here, we consider the steps required for *one* such computation.

Figure 1 illustrates the data-flow for two input images i and j to the final value to populate the matrix at (i, j).

symbol	description
P	extracts the sensor sensitivity values for each pixel in the image
Ff	'fast Fourier (forward)' transforms the matrix from space domain to frequency domain
R	rotates the matrix 180 degrees
x	creates a new matrix which is the cross-product of the inputs
Fb	'fast Fourier (backward)' transforms the matrix from frequency to space domain
e	extracts the Peak Correlation Energy value from the matrix

Naively, with infinite worker threads and infinite memory, the given information already provides the majority of the solver's implementation; for every pair, compute the correlation value and write it into the matrix. Already at this high-level, the data-flow

diagram already highlights an opportunity for speedup within a single correlation-computation; the 'jpg preprocessing' steps ($[P, Ff, R]$ for one and $[P, Ff]$ for the other) can be performed concurrently. With this, the runtime of the solver would be just $max(time(P, Ff, R), time(P, Ff)) + time(x, Fb, e)$. No CISI solver can beat this runtime, as these processes have unavoidable data-dependencies that prevent further concurrency.

Moving from the idealized world to to reality introduces a hindrance to our naively-parallel approach; we don't always have $N \times N$ workers available. Inevitably, for some large problem, some correlations must be computed *after* others. Fortunately, owing to the *all-to-all* nature of the CISI problem, many correlation values begin with *redundant* jpg preprocessing steps. The consequence of this observation is explored in §2.2.1.

2.1.2 Dependency-imposed Constraints

§2.1.1 identified a data-dependency between coarse tasks that manifest at an *algorithmic* level; namely, the independent and combined computations of two jpgs to one correlation value. However, owing to properties of some of our source-code dependencies, particular invocations on conceptually-unrelated data must be serialized. These dependencies are explained in the context of processes as they are outlined in §2.1:

- P requires access to a 'prnu-state' object to perform its work. Each such state must never be invoked by two threads at once. Additionally, per machine, there can only be one prnu-state being created or destroyed at a time.
- Ff and Fb each require their own 'fft-state' objects to perform their work. Each such object cannot be accessed in parallel, nor can any machine create or destroy more than one at a time. Additionally, a fft-state is irrevocably coupled to a provided matrix at creation-time, and it is on this matrix that it performs the transformation when invoked; this necessitates that the matrix 'outlives' the state object, and that no two matrices (in memory) can ever share an fft-state, even if they sequentialize their accesses.

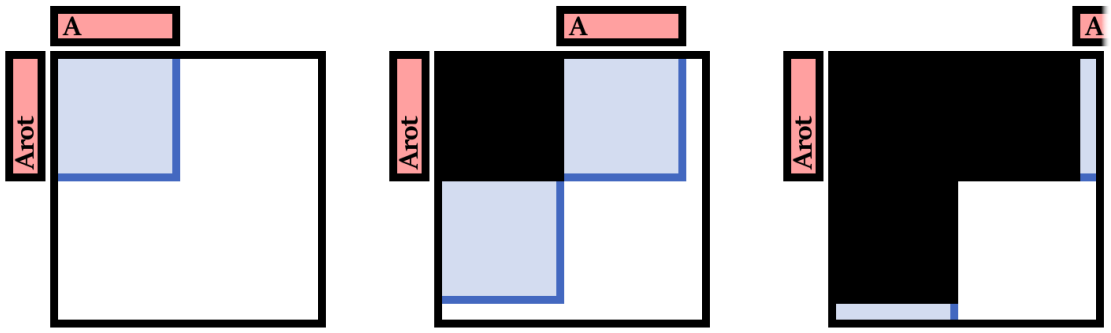
These restrictions are a result of the underlying thread-unsafe libraries in Chapel and C. On their own, these constraints are significant obstacles to concurrency. Coupled with the observation that every interaction with either type of state object takes *significant* time (a few seconds), minimizing their sequential computation time becomes the driving force behind the optimization process, explored in the section to follow.

2.2 Solver Optimizations

§2.1 provides enough to build a correct CISI solver. This section explores optimizations to achieve good performance in addition to correctness.

2.2.1 Caching

§2.1.1 observed the value of re-using *preprocessed jpgs* for correlation computations. Here we begin by considering another idealized circumstance, but one step closer to



(a) diagonal block **(b)** first edge block **(c)** second edge block

Figure 2: Steps taken by the optimized CISI solver when it's cache cannot contain all the needed intermediate jpg values. It first populates the (a) diagonal sub-array with symmetric axes, and then steps along the edge in blocks until the top rows and left columns have been populated with steps (b), (c) and so forth.

reality: infinite memory but finite workers. With computation time as the bottleneck resource, the best strategy becomes to traverse, precompute and store the intermediate representations[1] of every jpg in $\{1, 2, ..., N\}$. This produces a two-phase solver that must (1) populate the cache and then (2) compute correlation values.

This design is a step in the right direction, but requires an unrealistic assumption: infinite memory. Instead, we accept the fact that avoiding redundant jpg precomputations entirely is impossible. Instead, we seek to minimize recomputations with a cache limited to a maximum of 'm' slots, where m is a configuration parameter[2].

To populate a given $T \times T$ matrix, we begin the divide-and-conquer approach that needs no more that m cache slots. It begins by creating a *sub-task* to solve the first Q columns and rows, where $2 \cdot Q \leq m$. First, the cache is filled with Q rotated and Q non-rotated entries. The $Q \times Q$ matrix in the top-leftmost corner is then traversed in parallel, with workers accessing the cache as needed, computing and writing correlation values in-place. This is shown in Figure 2a. In the event that $Q = T$, the CISI instance has been solved. Otherwise, the latter $T - Q$ rows and columns remain unpopulated, but are not a convenient shape for the solver to work with. Rather, the solver divides-and-conquers yet again. The non-rotated values in the cache are overwritten with W new non-rotated precomputed jpg entries, where $Q + W = T$ and $W/2 \leq m$. The cache then contains the data needed to populate a $Q \times W$ block adjacent to the $Q \times Q$ block. This process repeats, walking along the edge, each time re-using the same Q rotated values in the cache with newly-loaded un-rotated values, shown in Figures 2b and 2c.

[1] Unfortunately, for a correlation we require one rotated, and one non-rotated jpg in a pair. Once preprocessed fully (with process *Ff* applied), rotation cannot easily be applied or undone, as *Ff* is not cheaply invertible. This necessitates that *two* intermediate representations be stored per image: one of which was rotated.

[2] This value is parameterized as memory allocation varies greatly depending on the run conditions. It will be seen later how selecting the largest possible value for m is of great benefit

Eventually, precisely Q rows and Q columns have been populated from T. The problem then recurses, treating the untouched $(T - Q) \times (T - Q)$ corner of $T \times T$ as the new, smaller matrix to populate.

2.3 Considering Task Parallelism

§2.2.1 describes how, with matrices large enough that $N > m/2$, the cache is insufficient for eliminating redundancy in precomputing jpgs entirely. Intermediate experiments showed that precomputing a single jpg takes approximately 3.7 times longer than computing the correlation between two cached jpgs. Ultimately, arbitrarily-large matrices necessitate any number of these sub-tasks, alternating between phases of (1) preparing jpgs and (2) computing correlation values.

At first glance, one may be tempted to break the data dependency between these phases such that some *latency hiding* may be performed: can we load images and compute them at the same time? Indeed, any number of viable schemes provide ways of overlapping these two phases repeatedly: dynamically dispatching workers, creating more fine-grained tasks, or partitioning the cache into many blocks for heterogeneous worker tasks at once. Ultimately, the problem of limited cache size necessitates barriers *somewhere*. Such barriers always introduce a risk of exacerbating load-imbalance; what if precompute-workers end up waiting for compute-workers or vice-versa? This load imbalance can be mitigated in the case of CISI, as the relative runtimes of the heterogeneous tasks are known. Conceptually, two very large tasks (one 'compute' and the other 'precompute') set upon by two static sets of workers will finish simultaneously if the ratio of tasks-to-workers is 3.7 larger in the former than the latter.

Clearly, such task-parallelism is only competitive[3] when there are *very many tasks*, as it smooths out any rounding problems, and a ratio of 3.7 cannot be approached otherwise. This requirement is the very same that renders the task parallelism useless; if there is *so* much work per worker, doing them as balanced sequential tasks will be comparably efficient[4]. To make matters worse, reliance on some expectation of relative runtime is sensitive to run conditions. Another environment might experience a runtime ratio closer to 4.7, but be optimized for 3.7 and thus suffer load-imbalance. The proverbial 'nail in the coffin' for this idea comes with the realization that overlapping compute with recompute would further sub-divide the cache such that each task has an (effectively) smaller cache-space to work with. The quadratic relationship between cache slots and the number of computations that can benefit from cache slots per task should make clear why shrinking the cache is very undesirable.

Ultimately, the final implementation does not overlap 'compute' and 'precompute' tasks. The only example of intra-task concurrency occurs when one jpg produces two

[3] We also disregard any additional eternal factors such as shared resources between workers. Such things of course further complicate matters. In our case, workers and cache slots can be provided with their own *prnu-state* and *fft-state* objects to avoid such problems.

[4] For problems with many workers and few tasks, dynamically allocating work would become useful, as it can potentially reduce worker idle time. We disregard this option as we anticipate large numbers of 'compute' and 'precompute' tasks per worker, owing to the largeness of expected matrices.

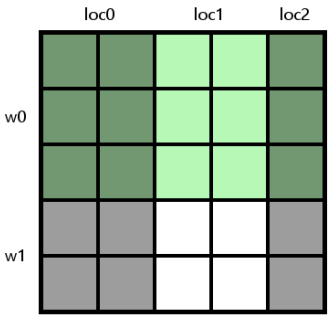


Figure 3: An example of a 5×5 matrix partitioning columns over three locales and rows over two workers per locale. Each cell represents a parallelizable task.

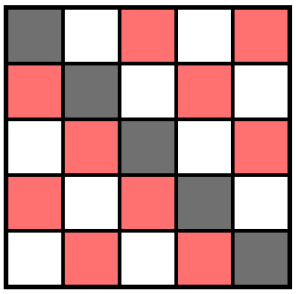


Figure 4: A mask for diagonal matrices (here, 5×5 marking that some cells can be skipped (shown in red) as their peer over the diagonal will be computed.

intermediate values to be cached. The rotated variant's work forks off part-way such that *Ff* can be computed in two places in parallel.

2.3.1 Data Parallelism

All parallel variants of the CISI solver explored in this work encounter tasks on large arrays ('precompute into cache slots 1 to 10') or matrices ('populate this matrix using cache slots 1..10'). In either case, each *locale* (Chapel's notion of a node in the distributed system) has a set of local *workers*, together comprising a global pool of workers. 'Worker' is an application-level concept that does not necessarily map 1-to-1 to threads on a locale. Rather, a *worker* with id 5 can be thought of as the only thread that has permission to access the *prnu-state* in slot 5. The necessity of this is explained in §2.1.2.

A data-parallelizable task is partitioned in approximately equal-sized blocks[5] in a manner depending on the circumstance such that all available workers end up with roughly equal loads. Figure 3 illustrates the case of populating the correlation matrix in a data-parallel fashion by partitioning columns over locales and rows over local workers. Note that partitioning over locales serves as a simple mechanism for maximizing data-locality; matrix rows correspond with jpg indices, which in turn correspond with cache slots. Workers can be sure that at least 1 of 2 of the cache slots can be accessed locally.

Population of a block of the correlation matrix that lies on the *diagonal* afford exploitation of the redundancy of their values. Recall that for such a block, as in the entire matrix, the correlation value of i with j is symmetric for all i and j. Figure 4 shows the '*work mask*' for these blocks, interpreted as cells for which the value does not need to be computed, assuming that when it's peer-value is computed, both cells are populated. This mask can be cheaply computed by each worker independently. It

[5] Cache slots, and partitioning schemes that operate on them which wish to maximize locality are instead distributed *cyclically*. This choice is mostly to simplify the source code as cyclical partitioning makes following the mapping of cache-slots to jpg-indexes easier. This deviation does not have a large impact on the ideas presented in this section.

essentially translates the *triangular* matrix into a *sparse* one such that work partitioned per-row or per-column ends up approximately load-balanced.

2.4 Final Design

The final, optimized, parallel and distributed CISI solver implementation attempts to leverage the observations made in the previous sections. It is fundamentally based on the following **observations** (some of which are assumptions):

1. Each node is able to cache up to m (a configuration parameter) intermediate jpgs without running out of memory. In most cases, $m < N$. Additionally, $m > 2$, and tends to be non-trivially-small (eg: we expect a dozen or two per node at least).

2. We wish to minimize barriers as they incur overhead and can magnify the downsides of load-imbalance. However, we acknowledge that with well-balanced loads and coarse tasks, their overhead is affordable.

3. Each node can support up to w (a configuration parameter) workers effectively.

4. We expect that each worker can safely allocate a small constant number of intermediate-jpg sizes at a time without running out of memory. This is necessary for workers to make local temporary copies of remote intermediate jpgs.

5. Loading and preparing a jpg (Processes $[P, Ff, R]$ in Figure 1) takes approximately 3 to 4 times longer than computing a correlation from intermediate jpgs (Processes $[x, Fb, e]$).

3 Experiments

This section explores experimentation on the optimized CISI solver as well as its variants.

3.1 Experimental Design

Evaluation of the implementations was conducted using the Distributed Asci Supercomputer (mark 4) cluster at the Vrije Universiteit Amsterdam. It offers dual quad-core worker nodes, each with 2.4 GHz processors, 24 GB of memory, networked using 1 Gbit/s bandwidth links. Processes were run *solo* on DAS-4 worker nodes.

All runtimes in this section omit time taken to initialize the state-objects *prnu-state* and *fft-state* (described in §2.1.2). Allocation of static variables such as the cache for the optimized solver are also not included.

3.2 Solver Variants

Ultimately, this project aims to produce a parallel and distributed solver well-optimized for the CISI problem. However, in this section we identify four distinct solvers that approximately represent various degrees of progress toward the optimized solver. Owing

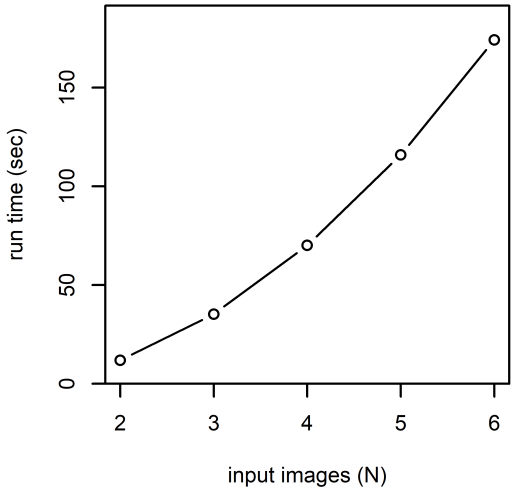


Figure 5: S_S runtimes for various small choices of N.

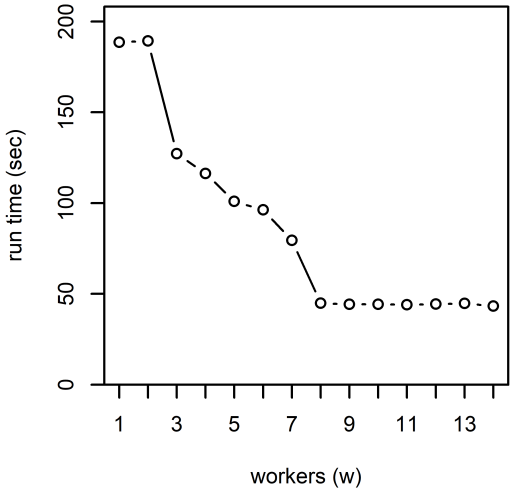


Figure 6: S_L runtimes with $N = 6$ for various choices of w.

to their differing degrees of complexity, the solvers also differ in the parameters they expose to users. This section explains the solvers, how they differ from one another, and which parameters are relevant to them.

name	parameters	description
S_S		Sequential solver on one thread.
S_L	w	Single-locale 'local' solver with w worker threads.
S_D	w, l	Distributed solver using l locales, each with w worker threads.
S_O	w, l, c	Optimized solver using l locales, each with w worker threads, and c cache slots per locale for storing intermediate jpgs. This is the solver defined in §2.4.

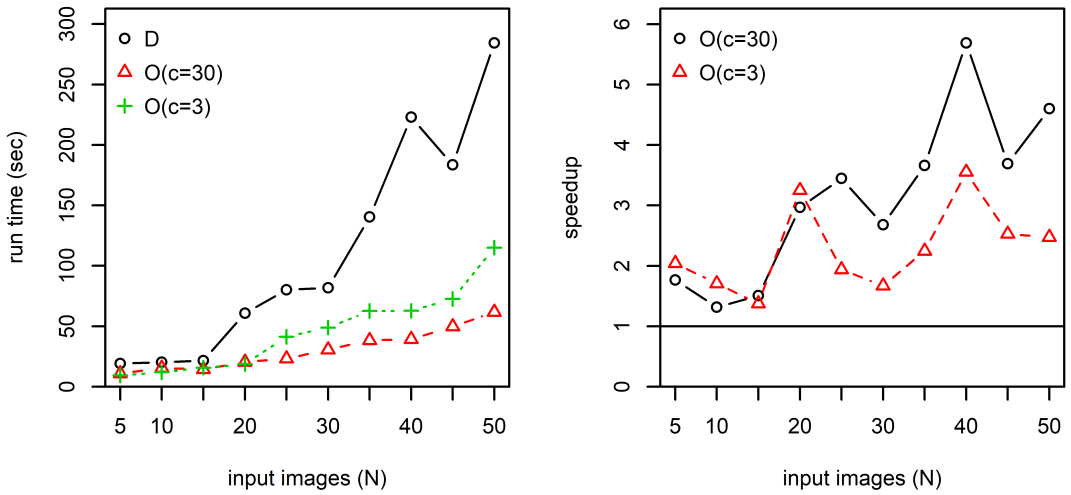
Note that in previous sections, the number of *total* cache slots, 'm', was more useful for reasoning. However, from the user's point of view, the restrictions in cache size tend to present themselves per locale. Here we refer to c such that $m = l \cdot c$ to avoid confusion.

In addition to the various configuration parameters defined above, 'N' (the number of input images, associated to 'problem size') is also provided per runtime measurements in this section.

3.3 Results

The runtimes seen in Figure 5 give a frame of reference for the most naive runtimes encountered when solving a CISI problem instance. Here, we see that small correlation matrices take considerable time to populate.

Matters are much improved when exploiting parallelism, even if done naively as seen



(a) Absolute runtimes

(b) Speedup of S_O runs with respect to S_L.

Figure 7: Runtimes for intermediate-sized problems of various size (N) for solvers S_D, S_O with $c = 30$ and S_O with $c = 3$. All solvers used $w = 8, l = 16$.

in Figure 6. Here, S_L partitions the matrix into rows, such that each thread can work independently. As described in §2.3.1, work is approximately balanced between these rows by each thread making independent use of the 'work mask'. The figure also shows that w values exceeding 8 had no beneficial effects on the runtime. Beyond this threshold, no thread has more than one row to traverse. The addition of more workers only increases the set of workers given zero rows. Indeed, in it's current form, S_L does not leverage additional workers by, for instance, partitioning the columns as well. Note that the speedup in the range of $\{3, 4, ..., 7\}$ is somewhat 'bumpy' or 'jagged'. For this small problem instance, the loads of each row are somewhat imbalanced, owing to the nature of the 'work mask'. The scheme by which workers are assigned rows has the effect of sometimes changing *which* row(s) a worker is assigned if the number of workers changes. This results in the irregular runtimes. However, the lack of speedup from 1 to 2 workers is harder to explain, but is most likely a result of how the Chapel language copes with a *coforall* loop with just one iteration. This case requires further investigation.

Figure 7 shows the advantages enjoyed by the distributed solvers S_D and S_O. The work partitioning pattern once again produces 'jagged' speedup in response to the problem size. In this case, workers are ultimately each assigned a rectangular blocks of the matrix (owing to block-partitioning per *locale* in one dimension, and *per thread on each locale* in the other). As before, slight shifts in alignment of these allocations have noticeable effects on the runtimes of the solvers. We are also able to observe an indisputable speedup of the optimized solver over its naive peer. As expected, this speedup is great, but still fundamentally restricted by the unavoidable cost of process Fb being necessary per pair-computation. We also are able to observe that the different choice of c for the S_O

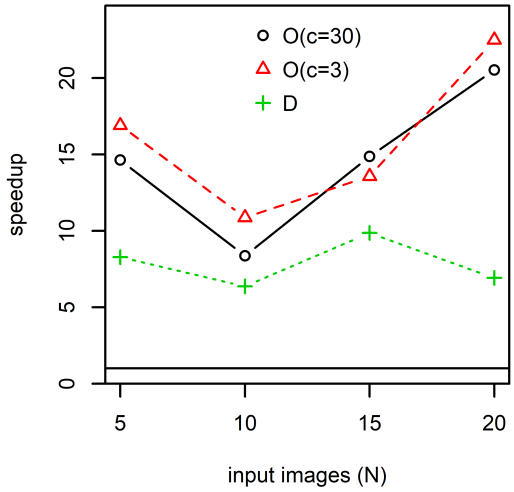


Figure 8: Results of $N = \{5, 10, 15, 20\}$ in Figure 7, plotted as speedup with respect to S_L runs using $w = 8$.

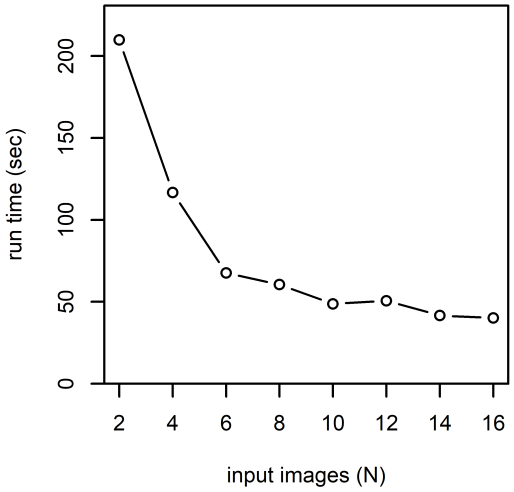


Figure 9: Runtimes of S_O with c=30, N=40, and w=8, for various choices of l.

solvers has no effect until the problem size is large enough. For N in $\{25, 30, ...45\}$ The problem size necessitated the division of the matrix into sequential sub-problems, as the necessary intermediate jpgs could no longer fit in the cache. For $N = 50$, a *third* compute task became necessary.

While clearly the more parallelizable solvers S_D and S_O shine for larger problem isntances, Figure 8 shows their superiority even for small problems in comparison to S_L. The optimized solver is the best of all, but at these scales the choice of c is irrelevant.

To test its scalability, S_O runs were performed with the same parameters, save for varying choices of l. Figure 9 shows that, as expected, more workers resulted in considerable speedup. As before, the runtimes are 'jagged' at small scales, a result of the 'work' mask interacting with the way blocks of the matrix are assigned to locale-specific workers. For $l = 2$, this CISI instance encountered additional slowdown as the cache (with total size proportional to l) became too small. However, thanks to well-balanced loads and no idle workers, the runtime for $l = 2$ is approximately at the expected level: twice that of $l = 4$.

4 Conclusion

The results in §3 paint the typical picture of an optimized solver for *any problem*: It's fast, but it also has developed certain quirks owing to the means by which is was optimized. The complex relationship of the 'work mask' and the block-like work partitioning among threads creates unintuitive variance in runtimes as a function of other properties of the CISI problem, namely, the number of images. in some cases, under otherwise identical

conditions, slightly larger problems are consistently solved faster! This variance suggests that tweaks to these work-distribution mechanisms have the potential to improve the solver's performance further. For example: would performance increase if a master thread partitioned matrix cells in a round-robin fashion to a global pool of workers? A user may also be surprised to learn of the extent to which the sufficiently-large choice of the c parameter is instrumental for performance. Fortunately, a reasonable compute cluster is able to support c values of sufficient size to see considerable speedup. Under ideal circumstances, the optimized solver is able to solve a 200-image CISI problem within 12.7 minutes, a feat inconceivable for the naive variants.

Chapel is a highly expressive programming language with many powerful mechanisms for expressing parallel tasks succinctly. The development of the implementation (once it had gotten off the ground) enjoyed rapid iteration and prototyping. This was owing to the language's ability to minimize coupling of work dispatch from other application logic. While it is still somewhat rough around the edges, the language was solid enough to facililate the development of a solver for a real-world problem in just a few weeks.