Parallel Programming Practical

Common Source Identification with Chapel

Trisha Anand (2596695)

1. Introduction

Common Image Source Identification computes within a set of images which images were made with the same camera. Due to the manufacturing process, not every pixel in a camera sensor has the same light sensitivity, a phenomenon that is named Photo Response Non Uniformity (PRNU). By applying a set of imaging filters we can extract this PRNU noise pattern that is unique for each camera. In this assignment we compute a correlation matrix with each value representing the correlation between a pair of images. To this end, we have used the Dresden database, a database with near identical images, each made with a different camera. The correlation matrix computed saves the peak to correlation energy metric (PCE) for image i and image j at index (i,j). This is implemented in partitioned global address space language Chapel in four versions : sequential, parallel (multi threaded) version running on a single locale, parallel version running parallely on multiple locales, and optimized parallel version running on multiple locales.

2. Design and Implementation

a. Sequential Single Naive

In the initialization phase, all the images file names are read. The dimensions of the first image in the given directory is saved and it is assumed that all the images in the directory are of the same dimensions as the first one. The following four domains are created to represent different data types :

- *corrDomain* : {1..n, 1..n} : This domain represents the correlation matrix where at index (i,j) PCE would be stored for image i and image j.
- *imageDomain* : {0..#h, 0..#w} : This domain represents image. Each image array with different data structures are stored using this domain.
- *numDomain* : {1..n} : This domain represents the range of number of images.
- *crossDomain* : {1.. #(n * (n-1)/2)} : Since PCE for image i and j is equal to PCE of image j and image i, one needs to only compute upper/lower triangle of the correlation matrix. This leads to the total computations being n*(n-1)/2. crossDomain represents the total work that needs to be done where each entry in crossDomain represents one PCE that needs to be computed.

To compute PRNU noise pattern of an image, one first needs to initialize the PRNU data structure. Since this data structure does not change, for each image's PRNU noise pattern computation, one need not be computing *prnu_init* every time. So *prnu_init* is part of initialization. Similarly once the noise pattern has been computed, one needs to apply a forward fourier transformation to this noise pattern and the rotated noise pattern. The *plan_dft* function has to be called once before computing for initialization. These fourier transformation initializations are also part of the init.

In the timed portion of the code, for each correlation that needs to be computed the following steps are adhered to :

- Read image A and image B into memory
- Calculate PRNU noise patterns for images A and B.
- Rotate PRNU noise pattern for image B.
- Calculate forward fourier transformation for image A and rotated image B noise patterns.
- Compute dot product of these fourier transformed matrices and save it in result matrix.
- Execute inverse fourier transformation on the result matrix.
- Compute the peak correlation energy metric from the resultant result matrix and save this value at index (A,B) in the correlation matrix.

After all the correlations are computed, the fourier plans and the prnu plans are destroyed. The correlation matrix is written to a file (if specified as a run time parameter) and the program exits.

b. Parallel Naive on a Single Locale

To parallelize the previous version, parallel independent work was introduced with the concept of 'threads'. All the fourier transformations and the prnu plans use a single data structure which doesn't allow parallel execution of prnu calculation and fourier transformations using the same data structure which has been initialized. To parallelize the timed step, the fourier plans and the prnu data are replicated 'x' times. Now the prnu and fft transformations can be parallelized at the maximum 'x' times.

Class ThreadData is introduced. ThreadData represents the work that would be executed serially. So ThreadData contains the fourier plans and the PRNU data structure. Number of objects of class ThreadData created during execution are configuration dependent (--*numThreads*) and represents the maximum possible parallelization.

In the initialization, instead of *plan_dft()* or *prnu_init()* being called only once, we create *numThreads* ThreadData objects and initialize the fft plans and prnu data structures once per ThreadData object. In this phase, each ThreadData object is also associated with the indices (low, high) in *crossDomain* representing the correlations that the particular 'thread' is responsible for.

In timed step, 'coforall' is issued to create *numThreads* tasks. In each task, corresponding ThreadData object is used for the fourier plans and prnu data strucutre. Each task calculated the PCE for the indices in crossDomain that were allocated to it during initialization. The PCE calculation steps remain the same as that of sequential.

c. Parallel Naive on Multiple Locales

To run the previous version on multiple locales, *crossDomain* was Block mapped.

```
var nrCorrelations = (n * (n - 1)) / 2;
var crossDomain = {0..#nrCorrelations} dmapped Block({0..#nrCorrelations});
```

This led to all the correlations that need to be computed to be distributed equally to all the locales. Now each locale needs to only compute subdomain of *crossDomain* which has been allocated to it by Block() construct.

On each locale, ThreadData objects are created according to the config variable *numThreads*. In the initialization, the indices that each threadData gets associated to is now given from the subdomain of *crossDomain* allocated to a particular Locale instead of the entire *crossDomain* as was the case with the previous version.

The timed step remains the same as the previous version.

d. Parallel Optimized on Multiple Locales

i. Optimization 1 : Implicit type casting

The array of prnu of data type real need to be converted to an array of complex numbers. In the naive version this is done as follows :

```
forall (i,j) in imageDomain {
   prnuComplex(i,j) = prnu(i,j) + 0i;
}
```

This was changed to implicit type casting instead of assigning each individual element.

```
prnuComplex = prnu;
```

ii. Optimization 2 : Localization

The Chapel Visualizer was used to understand the implicit and explicit communication that happens between locales. One of the implicit communication happening very often was communicating values ‘h’ and ‘w’ (denoting the height and width dimensions of the images) from Locale 0 to all the other Locales. These dimensions were communicated multiple times during each PCE computation. To overcome this, on each Locale local *h* and local *w* was introduced (*h_loc, w_loc)*. They are communicated only once to each Locale and then only the local values are used henceforth.

iii. Optimization 3 : Implicit to Explicit Parallelism

Chapel’s supports implicit data parallelism when we do operations like C = A + B where A, B and C are arrays. This was used in a lot of places like calculation of dot product of the noise pattern arrays (*C* = *A***B*) or sum of all the elements (+ reduce *array*), etc. This was changed into explicit data parallelism using *forall* loop. This led to a marked improvement in performance in some scenarios.

iv. Optimization 4 : Cache

The next optimization added was in memory cache. The idea behind this was to decrease the computations required. On each Locale two arrays are introduced to store noise patterns with forward fourier transform applied. One array caches the rotated version and the other array caches the non rotated matrices. The caching exists at Locale level allowing all the threads to access, write to and if available read from the cache. The size of these arrays are a configuration variable and can be changed using --*maxCache* variable at runtime.

```
var cachePrnuIdx, cachePrnuRotIdx  : [{0..#maxCache}] int;
var cachePrnu : [{0..#maxCache}][imageDomain] complex;
var cachePrnuRot : [{0..#maxCache}][imageDomain] complex;
```

cachePrnu and *cachePrnuRot* are the actual arrays that are used to implement the cache functionality. These arrays store the noise patterns on which the fourier transformations have been applied. *cachePrnuIdx* and *cachePrnuRotIdx* are the two arrays which are only used to store the indexes of the images whose matrices are stored in the *cachePrnu* and *cachePrnuRot*. If an image A exists in *cachePrnu* array at index ‘idx’, then its guaranteed that *cachePrnuIdx(idx)* would store A.

By design the cache arrays only store the first *maxCache* matrices. Once the cache is full, no more writes happen and the cache arrays are only used to read.

Since no mutex locks exist in Chapel, atomic boolean variable is used to introduce single access to writing to the cache array.

```
lock : atomic bool
var (found, idxVal) = cacheIdx.find(i);
   if (!found) {
     while lock.testAndSet() do chpl_task_yield();

     cache[cacheSize.read()] = val;
     cacheIdx[cacheSize.read()] = i;
     cacheSize.add(1);

     lock.clear();
   }
```

Now before reading an image into memory, first a check is made in the cache and if the cache returns a positive result, we skip directly to the dot product step.

3. Run Time Measurements

The following experiments have been run on DAS 4 with varying number of machines. The graphs have been used to present the data. The raw data for each of the figures can be found in Appendix at the end of the assignment.

a. Performance of different versions

On 10 images, all the four versions of code were run to compare against each other. Figure 1 shows the speedup of all four versions against the seq-single-naive. The raw data for this graph can be found in Table 1 in Appendix.

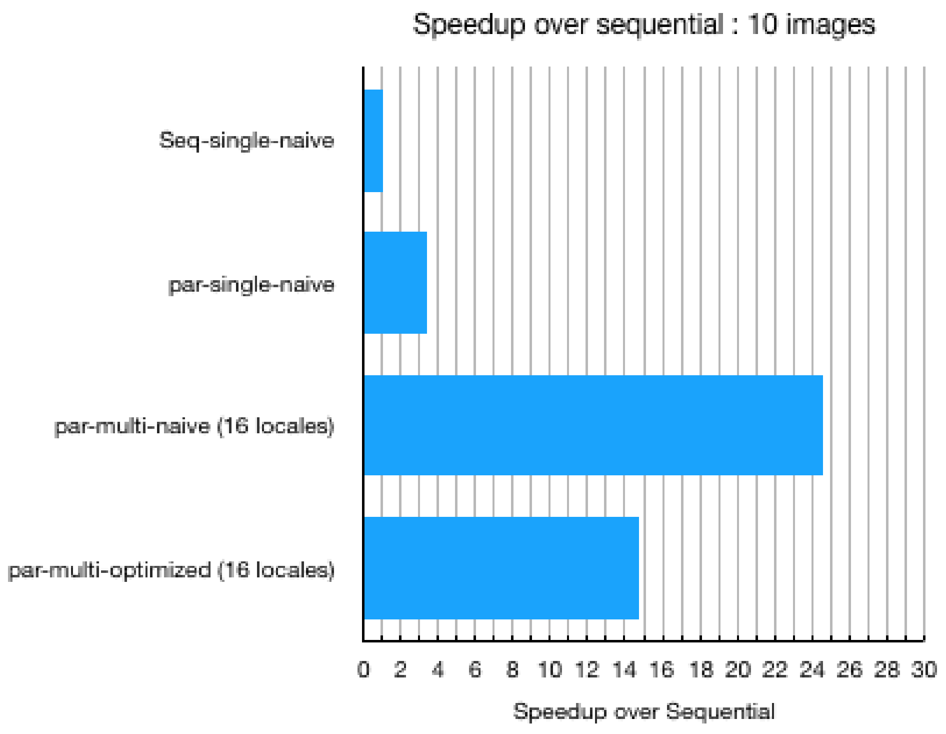

Figure 1 : Speedup of four versions on 10 images over seq-single-naive

As can be seen in Figure 1, the performance is increasing as we move from seq-single-naive to par-multi-naive over 16 locales. The reason we are observing a drop of performance from par-multi-naive to par-multi-optimized is because the problem size is too small to derive any benefit from caching but suffers from all the extra work that needs to be done for caching.

b. Optimization Application

The performance was measured across different number of Locales while applying the optimizations mentioned in Section 2 on the par-multi-naive version. Figure 2 shows the performance for the four different optimizations applied for the problem size of 20 images. The raw data for this can be found in Table 2 in Appendix.

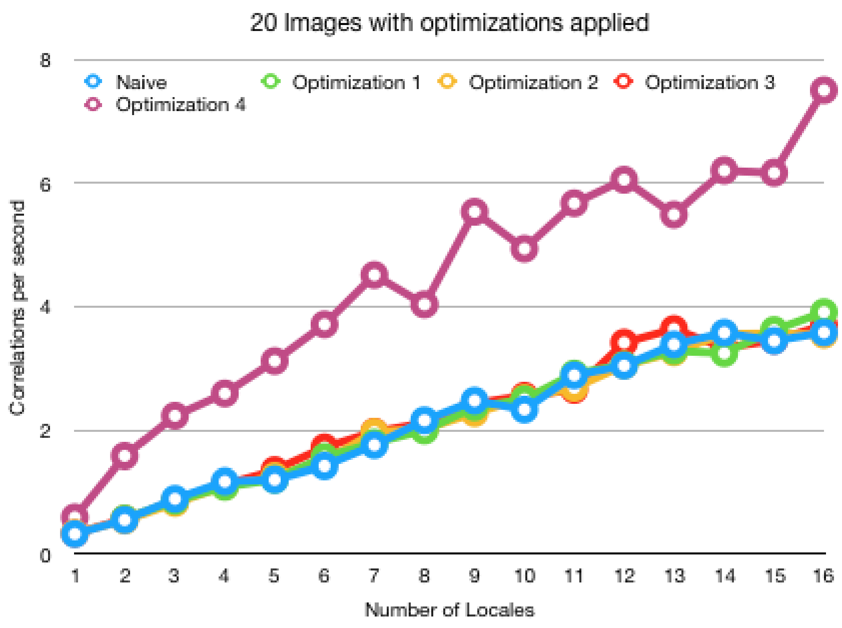

Figure 2 : Optimizations applied on problem size of 20 Images

This shows that optimizations 1, 2, and 3 didn't impact the performance greatly whereas considerable speedup is achieved using optimization 4 which was caching.

c. Cache Behaviour with different problem sizes

To see the efficacy of the caching feature added as optimization 4, cache hits were recorded for two different problem sizes : 30 images and 50 images with the program running on 16 machines. In the figures below, 'cache hit PRNU' denotes finding noise pattern for an image on which forward fourier transformation has been applied. Similary 'cache hit PRNU Rotated' denotes noise pattern matrix which was rotated and applied fourier transformation on. Cache Hits denote the number of times a noise pattern matrix was already existing in the cache array and didn't need to be computed.

Figure 3 below shows the cache hit statistics on 30 images. The raw data for this can be found in Table 3 in Appendix.

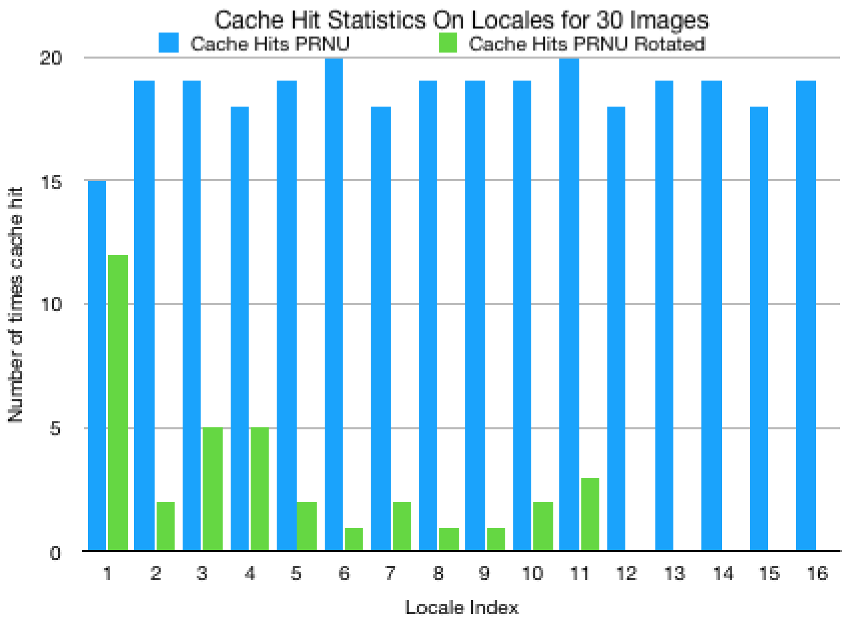

Figure 3 : Cache Hit Statistics on Locales for 30 Images

Figure 4 below shows the cache hit statistics on Locales for 50 images. The raw data for this can be found in Table 4 in Appendix.

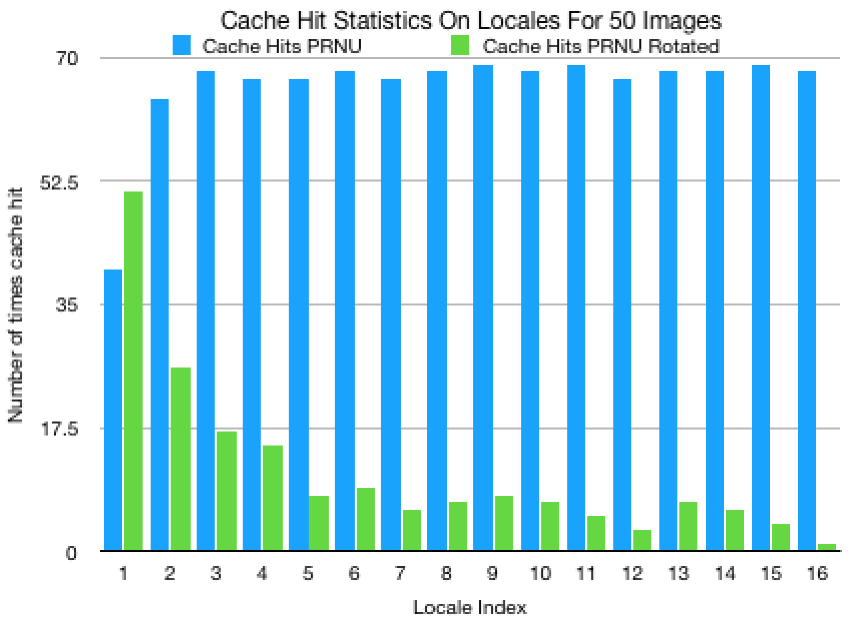

Figure 4 : Cache Hit Statistics on Locales for 50 Images

For 30 images, the performance was 10.2 corrs/sec and for 50 images, the performance achieved was 11.7 corrs/sec. The total number of correlations needed to be computed per locale for 30 images were 27 correlations per locale. Total number of correlations per locale for 50 images comes out to be 76. As can be observed from Figures 3 and 4, at least for PRNU noise pattern (and not the rotated one) a very high cache hit is being achieved and for most PRNU noise patterns, the computation is being done only once and used multiple times. Unfortunately for the rotated noise pattern, the cache hits at higher index is abysmal. As the problem size increases (30 to 50) the cache hits for the rotated noise pattern do increase leading to better performance.

d. Varying Problem Size

The optimized version was run with varying problem sizes to study the change in performance achieved. Figure 5 shows the chart representing the performance studied as correlations/second achieved. The raw data for this can be found in Table 5 in Appendix.

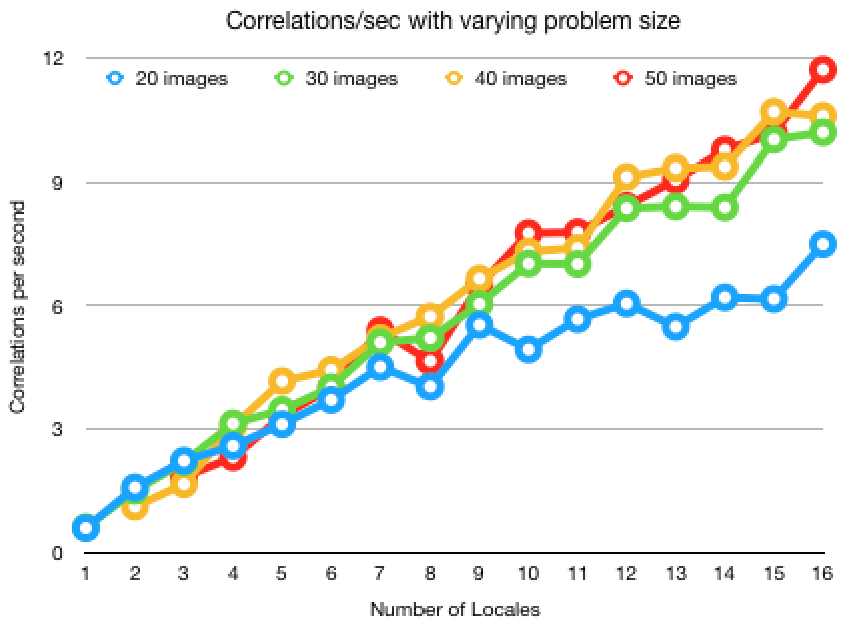

Figure 5 : Correlations per second achieved with different problem sizes

As can be seen in Figure 5, the performance improves with increasing the computations per locale (by increasing the problem size) when using a higher number of locales. Since the cache size is limited and the cache is fixed once its filled up, when using lower number of locales for the problem similar performance is observed for different problem sizes. As the number of locales increase, the cache contents are more relevant to the correlations being computed in the locale.

To overcome this, different replacement strategies should be applied to better the performance. To make Rotated PRNU cache more relevant or to balance the cache hits between PRNU and Rotated PRNU, the correlations being distributed to each locale could be different. This can be explored in future work.

4. Conclusion

As part of this assignment, Common Source Identification problem was iteratively implemented in Chapel language. The aim of the assignment was to achieve the best possible speedup from the sequential version. The best performance achieved using Chapel was 11.7 corrs/sec. Different optimizations are discussed and their impact on the performance have been discussed in experiments.

5. Appendix

This section contains all the data from which the figures in this assignment have been created.

a. Table 1 : Performance comparison of four versions

Table 1 below lists the performance achieved in correlations per seconds for 10 images of four versions of the chapel program

Version	**Performance (Corrs/sec)**	**Speedup over Seq-single-naive**
Seq-single-naive	0.0886441	1.00
par-single-naive	0.298672	3.37
par-multi-naive (16 locales)	2.1763	24.55
par-multi-optimized (16 locales)	1.3127	14.81

b. Table 2 : Optimizations applied on 20 Images

Table 2 below performance achieved in correlations per seconds for varying number of Locales with different optimizations as covered in section 2.

NumLocales	Naive	Optimization 1	Optimization 2	Optimization 3	Optimization 4
1	0.330437	0.331098	0.346144	0.339069	0.600305
2	0.557247	0.581234	0.547745	0.57372	1.58756
3	0.899325	0.866985	0.832918	0.857838	2.23621
4	1.17946	1.09254	1.10357	1.13156	2.59763
5	1.20755	1.21031	1.26243	1.35798	3.12271
6	1.42764	1.57175	1.50122	1.72578	3.7158
7	1.7616	1.80121	1.97352	1.97753	4.51593
8	2.162	1.99756	2.02563	2.11797	4.03959

9	2.48137	2.37428	2.27915	2.43755	5.53878
10	2.33665	2.5044	2.52235	2.56053	4.94098
11	2.89015	2.91962	2.68804	2.6553	5.67715
12	3.04769	3.06249	3.05289	3.4205	6.0579
13	3.39071	3.2871	3.26433	3.64175	5.49275
14	3.58057	3.24873	3.56107	3.33777	6.20588
15	3.44915	3.62522	3.56987	3.45514	6.16835
16	3.58468	3.91293	3.54757	3.70283	7.50001

c. Table 3 : Cache Hits Statistics On Locales : 30 images

When the program was executed over 16 locales with problem size of 30 images, the following statistics were recorded for cache usage as shown in Table 3 :

Locale ID	Cache Hits PRNU	Cache Hits PRNU Rotated
1	15	12
2	19	2
3	19	5
4	18	5
5	19	2
6	20	1
7	18	2
8	19	1
9	19	1
10	19	2
11	20	3
12	18	0
13	19	0
14	19	0
15	18	0
16	19	0

d. Table 4 : Cache Hits Statistics On Locales : 50 images

When the program was executed over 16 locales with problem size of 50 images, the following statistics were recorded for cache usage as shown in Table 4 :

Locale ID	Cache Hits PRNU	Cache Hits PRNU Rotated
1	40	51
2	64	26
3	68	17
4	67	15
5	67	8
6	68	9
7	67	6
8	68	7
9	69	8
10	68	7
11	69	5
12	67	3
13	68	7
14	68	6
15	69	4
16	68	1

e. Table 5 : Correlations/sec with varying problem size

The following table 5 shows the correlations/sec achieved with different problem size when the application was run with different number of total locales used for computations. The boxes left empty are the ones which couldn't finish running in the 15 minutes time frame allocated for every job on DAS 4.

Num Locales	20 images	30 images	40 images	50 images
1	0.600305	0.620845	—————	—————

2	1.58756	1.50604	1.12099	————
3	2.23621	2.20101	1.66552	1.84136
4	2.59763	3.14207	3.0738	2.32
5	3.12271	3.46979	4.17146	3.32037
6	3.7158	4.02355	4.44374	4.00995
7	4.51593	5.11248	5.22391	5.40573
8	4.03959	5.21788	5.73953	4.66039
9	5.53878	6.04774	6.6625	6.50029
10	4.94098	7.02904	7.32323	7.76642
11	5.67715	7.01776	7.40626	7.79075
12	6.0579	8.37228	9.12771	8.4309
13	5.49275	8.41546	9.33895	9.05484
14	6.20588	8.3927	9.37648	9.77219
15	6.16835	10.0224	10.6939	10.1746
16	7.50001	10.1958	10.5931	11.7159