# Chapter 4

# Parallel Lossless Hyperspectral Image Compression

** The content of this chapter has been published by Yaman Dua, Vinod Kumar, and Ravi Shankar Singh. "Parallel lossless HSI compression based on RLS filter." Journal of Parallel and Distributed Computing 150 (2021): 60-68.* `https://doi.org/10.1016/j.jpdc.2020.12.004` *(Elsevier, IF: 3.734)*

## 4.1 Introduction

High-Performance Computing (HPC) is the demand of the latest technologies like image processing, IoT, deep learning, machine learning, artificial intelligence, block chain, real-time systems. These technologies require more processing elements (both CPU and Input/Ouput (I/O)) than traditional ones due to the time complexity associated with it and the availability of more data than it was available in history.

HPC enables the use of multiple processors/cores/threads for running advanced programs efficiently [151]. It involves implementation of application programs on Grid computers, Cluster Computers, Distributed computers, multi-core computers, GPUs [152], and FPGAs [153]. It requires two types of modification in sequential programming

technique, one at the hardware level, i.e., designing hardware to execute more than one process at a time. The second modification is required at the software level, i.e., reorganization of the program such that it can use the maximum processing power of the hardware. Hardware modifications are already going on, and systems with petaflops or $10^{15}$ floating-point operations per second are available. The focus of most of the researchers is to develop programs that can efficiently and reliably be executed on these systems.

There are two categories [154] of parallel computation: implicit parallelism, and explicit parallelism. Implicit parallelism is processed by the architecture of the compiler, operating system, and hardware that can't be controlled by the design of the algorithm. Explicit parallelism is carried out by developing an algorithm in such a way that it could take benefit from the available resources [155]. Figure 4.1 represents the different forms of parallelism that could be exploited in a broad application.

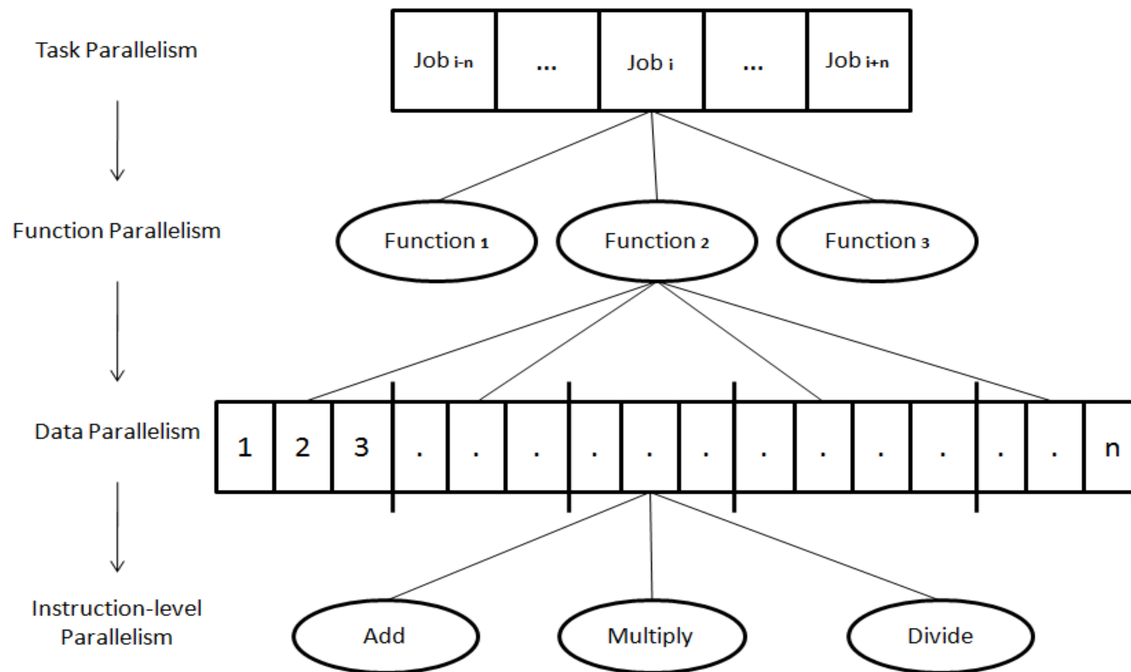


**Figure 4.1**: Identifying multiple level parallelism in an application

HPC is used by academic institutions, engineers, researchers, military, government

agencies, businesses of all sizes, and in the health sector. It primarily focuses on completing tasks faster (performance), and computations on remote devices with high reliability and scalability.

In remote sensing, apart from storage and transmission within Earth, HSI compression is mainly processed on satellite, and decompression is mainly processed on the data center, where it has to be processed in real-time. Failing which can result in asynchronous acquisition and transmission of images. Moreover, these algorithms suffer from large execution time due to a large number of pixels on which it has to process. HPC provides a mechanism to reduce the run time of such time-critical algorithms by dividing them into sub-problems and executing them simultaneously. Existing use cases of parallel computing [50] have already proven that satisfactory speedup can be obtained by the use of HPC paradigms on image processing applications. In this chapter, we modified existing prediction based algorithm to remove the dependency among variables. Identified optimum number of prediction bands for spectral decorrelation and number of pixels for spatial decorrelation. It is followed by parallel implementation on the supercomputer, PARAM SHIVAY (192 nodes, 40 cores per node, 384 GB per node DDR4 RAM).

**Parallel programming for high dimensional data**

Data collection and information extraction are the two most important steps of prediction based problems. Since data collection is a one-time process, recent developments mainly suggest optimization techniques for improving the accuracy in the information extraction step, including prediction models, neural network models, RNN models, CNN models [156]. Due to the high dimensional structure & size of data, and complexity of these techniques, it takes much time in computation that can be reduced by parallel programming. Parallel computing models have been used for image processing applications previously that permit the use of parallel models like the shared-memory model, distributed memory model, hybrid model and GPU use [28] in HSI compres-

sion. SMM [157] has been used for calculation of local features of an image [158], object identification in multi-spectral imaging [159], face detection using neural network [160], image block representation [161], distributed memory model [157,162–164], and hybrid model [165] used for image processing.

RLS filter was first used as a predictor in [147] that finds adaptive step-size of historical values. In [166], an improvised feature of forgetting factor was introduced to reduce the effect of previous pixels on the current predicted pixel value. C-RLS uses a large context window for spatial decorrelation achieving better results. To improve the prediction performance and reduce the run time of the CRLS algorithm, Karaca in [29] proposed a new algorithm named B-CRLS that uses spatial correlation to compress HSI further. In [167], a Fast-RLS-ALP algorithm was proposed to compress HSI on satellite using an RLS filter. Prediction based algorithm is also used in conjunction with other algorithms to improve performance. It is worth to mention that run time of RLS filter based prediction is more, but it has better performance than existing algorithms. In this work, a novel method has been proposed to reduce the running time of RLS based prediction algorithms by executing the algorithms on parallel computers.

## 4.2 Proposed Method

Parallel programming paradigms including data parallelism & task parallelism have been used to modify the existing RLS filter based prediction algorithms. Vectorization is also considered through programming, which handles instruction level parallelism. The algorithm works in two-phase, namely spatial decorrelation and spectral decorrelation. Spatial correlation in HSIs is available due to high spatial resolution resulting in similar values of a pixel as of neighboring pixels, which is removed by taking the local difference. Spectral correlation is present due to the higher resolution of imaging sensors & narrow, continuous and overlapping spectral bands.

**Recursive Least Square (RLS)**

RLS filter [147] works adaptively as an optimization problem to minimize weighted linear least square cost function by recursively updating the weight matrix. In HSI compression, it predicts the value of a pixel of the $n^{th}$ band from a vector of '$p$' collocated pixel values, where $p \epsilon [1, n-1]$ (both inclusive) is the prediction order.

The first step of the RLS algorithm is intra-band prediction, which focuses on spatial decorrelation in the spatial domain. Let $S(x, y, z)$ be the current value of a pixel at $(x, y)$ position on the $z^{th}$ band then,

$$D(x,y,z) = S(x,y,z) - \frac{S(x,y-1,z) + S(x-1,y,z) + S(x-1,y+1,z) + S(x-1,y-1,z)}{4} \tag{4.1}$$

The local difference matrix of a band '$z$' is calculated using equation 4.1 for all the pixels of that band. $D(x, y, z)$ depends on four previous values, as shown in Figure 4.2. The same process is repeated for all the bands of HSI, forming a 3-D local difference matrix of size $(H, W, Z)$, where $H$ = number of rows or height in HSI, $W$ = number of columns or width in HSI, $Z$ = total number of bands in HSI. The local difference matrix is used as an input signal to the second step of the RLS algorithm.

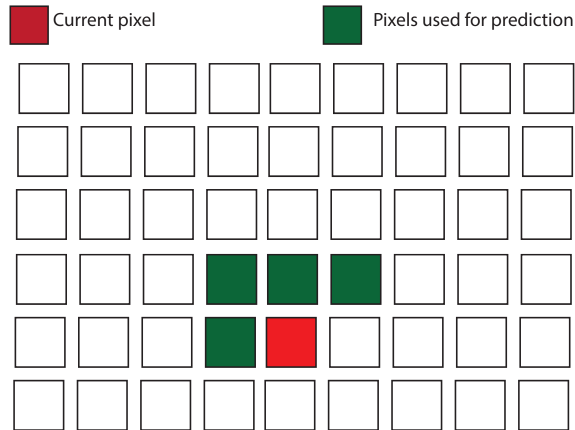


**Figure 4.2**: Neighboring pixels used for calculating the local difference

The second step is used to remove spectral decorrelation using the prediction residual. Residual values for each band $z \epsilon Z$ represents the difference in original local

difference value of pixels from the predicted value of that pixel, and this process is called inter-band prediction. Figure 4.3 represents the pixels used for the spectral decorrelation phase of RLS algorithm. In the figure, already predicted green colored pixels of '$n$' previous bands $(k-1, k-2, ..., k-n)$ are used to predict red colored pixel of $k^{th}$ band.

The computational complexity of Sequential RLS algorithm can be calculated by estimating the steps involved in the most critical phase, i.e., updating weight matrix. It consumes $O(p^2)$ time for each pixel in HSI, where $p$ is the prediction length.

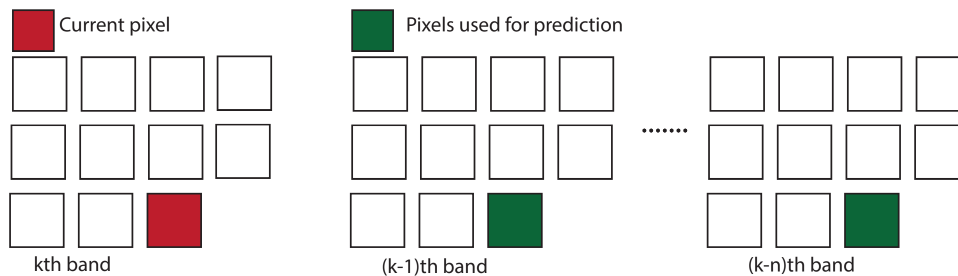


**Figure 4.3**: Pixels used for calculating the residual matrix

### Parallel implementation of RLS

By looking at the structure of given RLS algorithm for HSI compression, it can be observed that it is SIMD-type application, where the same function iterates on $Z$ number of bands that can be parallelized by the concept of data parallelism Further, algorithm 4.1 is examined for task parallelism or instruction parallelism and also converted to SIMD + MIMD-type application. The proposed algorithm is a modified version of [147] that can attain both versions of parallelism using a hybrid model.

### Conventional Recursive Least Square (A-CRLS)

The conventional RLS algorithm [166] differs from traditional RLS [147] only in the implementation phase. It uses a context window of 24 spatial pixels to perform spatial decorrelation. Figure 4.4 represents the context window of the CRLS algorithm. Equation 4.1 can be extended to more general form to calculate local difference matrix

**Algorithm 4.1:** Parallel-RLS Algorithm

```
   Result: Encoded bit-stream of prediction error
1  Input: Hyperspectral Image S of size H × W × Z, where H = number of rows,
   W = number of columns, Z = number of bands
2  Representation: d_z(t) = [d_{z-1}(t), d_{z-2}(t), ..., d_{z-p}(t)] is local difference input
   vector, w(t) = [w_1(t), ..., w_p(t)] is weight matrix, k^T(t) is kalman gain, and e_z(t)
   represents prediciton error
3  Initialization: Let P(0) = δI_p, w(0) = [0], t = 1, where δ = 0.0001 and I_p is the
   identity matrix of order p;
4  parallel for z = 1 to Z do
       // Executed in parallel on multiple number of processors
5      Calculate d_z(t) and d_{z-l}(t), where l = 1, 2, ..., p, and generate the input
           vector as d_z(t) = [d_{z-1}(t), d_{z-2}(t), ... , d_{z-p}(t)]; // Spatial
           Decorrelation
6  end
   // Wait till all the processors complete their job and return the
       value d_z(t)
7  parallel for z = 1 to Z do
       // Executed in parallel on multiple number of processors
8      parallel for height = 1 to H do
9          for width = 1 to W do
10             t = height · W + width; // location of pixel
11             Calculate the prediction error using
12             e_z(t) = d_z(t) − ⌊d_z(t) w^T(t − 1)⌋ ;
13             k^T(t) = P(t − 1) · d_z^T(t) / (1 + d_z(t) · P(t − 1) · d_z^T(t)); // Calculate gain
14             P(t) = P(t − 1) − k^T(t) · d_z(t) · P(t − 1);
15             w(t) = w(t − 1) + k(t) · e_z(t); // Update weight
16         end
17     end
18     Send the value e_z(t) to the adaptive arithmetic encoder for bit-code
       generation;
19 end
```

of CRLS algorithm, as equation 4.2 and 4.3:

$$\tilde{S}\left(x,y,z\right)=\frac{1}{24}\left(\sum_{i,j=0}^{3}S\left(x-i,y-j,z\right)\right) \tag{4.2}$$

$$D\left(x,y,z\right)=S\left(x,y,z\right)-\tilde{S}\left(x,y,z\right) \tag{4.3}$$

where, $\tilde{S}\left(x,y,z\right)$ represent reconstructed pixel value,$S\left(x,y,z\right)$ represent original pixel value, and $D\left(x,y,z\right)$ represent local difference value at location $\left(x,y,z\right)$ of HSI respectively. In Spectral-domain, prediction order (number of previous bands used for prediction) in the CRLS algorithm is obtained by performing a brute force method on the initial scene. The number of bands giving optimum results for this scene is selected as a prediction order for other scenes of that image. The computational complexity of sequential A-CRLS is explained in the literature [166], which has been significantly reduced by executing it on various processors in this work.

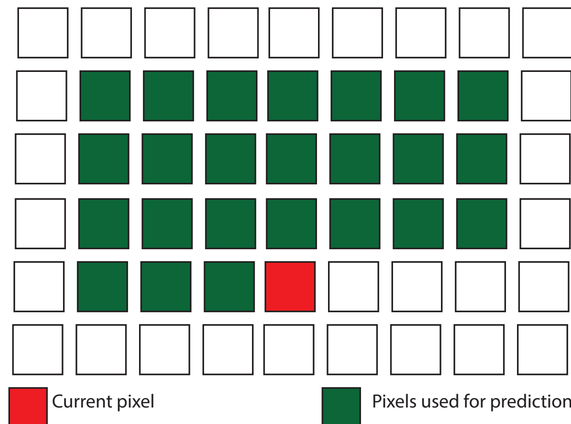


**Figure 4.4**: Pixels used for spatial decorrelation in A-CRLS

### Parallel implementation of A-CRLS

The algorithm is again Single Instruction Multiple Data (SIMD) type and can be converted to the SIMD + Multiple Instruction Multiple Data (MIMD) type application. The modified algorithm can then be implemented on a multi-core and multi-node processor as Algorithm 4.1 with a slight modification in parameter initialization.

### Recursive Least Square-Adaptive Length Prediction (RLS-ALP)

HSI compression using the RLS-ALP algorithm gives optimal results compared to the traditional RLS algorithm but is more time-consuming. RLS-ALP finds out optimum prediction order by executing the algorithm multiple times, changing prediction order from 10 to 200 in steps of 10. In the last step, it selects the prediction order giving the best results and executes the model. The computational complexity of sequential RLS-ALP is $O\left(t{\cdot}p^2\right)$, where $t$ is the number of times RLS algorithm is executed to find the optimal prediction length($p$).

### Parallel implementation of RLS-ALP

The algorithm for RLS-ALP can be modified according to the criteria discussed in Algorithm 4.2. In this case, task parallelism is found to be more critical than data parallelism by executing algorithms with different prediction orders on a different node, and each algorithm is executed in parallel on different cores of a node.

### Fast Recursive Least Square ALP (Fast-RLS-ALP)

A sequential algorithm for FAST-RLS-ALP is given in [167] that mathematically reduces the execution time of the RLS-ALP algorithm by changing the method to calculate the weight. It is optimal for on-board compression of BIL imagery. Parallel version given in Algorithm 4.2 is generated from the algorithm already proposed [167]. The computational complexity of sequential FAST-RLS-ALP is significantly reduced to $O\left(p\right)$, where $p$ is the prediction length, by replacing the multiplication step in weight calculation with a matrix append operation.

### Sequential implementation of algorithms

The sequential algorithms have been optimized through vectorization [168] using the coding paradigm of python. Vectorization is used to run multiple operations from a single instruction accomplishing the goals of SIMD architecture. It operates by utilizing most of the registers available with the system.

**Algorithm 4.2:** Parallel-Fast-RLS-ALP Algorithm

```
Result: Encoded bit-stream of prediction error
1  Input: Hyperspectral Image S of size H × W × Z, where H = number of rows,
   W = number of columns, Z = number of bands
2  Representation: d_z(t) = [d_{z-1}(t), d_{z-2}(t), ..., d_{z-p}(t)] is local difference input
   vector, w(t) = [w_1(t), ..., w_p(t)] is weight matrix, k_z(t) is kalman gain, and e_z(t)
   represents prediction error
3  Initialization: Let l = 3, c_z, k_z, w_z = 0, V_z = 1, P(0) = I_p (identity matrix of
   order p);
4  parallel for x = 1 to H do
       // Executed in parallel on multiple number of processors
5      parallel for y = 1 to W do
6          Obtain the mean value of pixel (x, y, z) using
7          Ŝ(x, y, z) = (S(x,y−1,z)+S(x−1,y,z)+S(x−1,y+1,z)+S(x−1,y−1,z))/4
8          Calculate d_{1,Z}(n) = S(n) − Ŝ(n); // Spatial Decorrelation
9          for each local difference d_Z(n) in d_{1,Z}(n) do
10             ϵ_Z(n) = d_Z(n) − round(d_{1,Z−1}(n) · w_Z(n)); // Calculates
                   prediction error matrix
               // Use RLS for initial l bands
11             if Z < 1 then
12                 Apply RLS to update P_Z(n) and k_Z(n)
13             end
14             Use Fast RLS in the form of append operation to matrix c_Z, which in
               turn calculates k_Z(n)
15             w_Z(n) = w_Z(n − 1) + k_Z(n) · ϵ_Z(n); // Update Weight matrix
16             Send ϵ_Z(n) to the adaptive arithmetic encoder for bit-code generation;
17         end
18     end
19 end
```

## 4.3 Experimental Results

Proposed parallel algorithms have been implemented to reduce the running time of algorithms using a parallel programming paradigm, i.e., shared memory parallelism. The following sub-sections contain detailed information about the dataset used, implementation environment, and simulation results.

**Dataset**

The algorithm was evaluated on the standard CCSDS [2] Hyperspectral dataset available in the public domain. It consists of 12 HSIs produced by the AVIRIS sensor, with the details given in Table 4.1. The number of rows is equal to 512 and the number of bands in each image is equal to 224, there are five images of the yellow stone scene that are calibrated and represented by YSC, the number of columns in these images is 677. Each pixel is represented using 16 signed bits, and scene number equal to 0, 3, 10, 11, 18. Figure 4.5 represents the grayscale image of the datasets.

**Table 4.1**: Description of dataset used in the experiment

| Dataset/ Attributes | Rows | Columns | Bands per image | Scenes |
|---|---|---|---|---|
| AVIRIS Calibrated | 512 | 677 | 224 | 0, 3, 10, 11, 18 |
| AVIRIS UnCalibrated | 512 | 680 | 224 | 0, 3, 10, 11, 18 |
| Hawaii | 512 | 680 | 224 | 10 |
| Maine | 512 | 614 | 224 | 1 |
| AVIRIS Calibrated | 512 | 677 | 224 | 0, 3, 10, 11, 18 |

Dataset also consists of 5 uncalibrated yellow stone images with scene numbers 0, 3, 10, 11, 18. The number of columns in these images equal to 680 with the 16 unsigned bits used to represent each pixel, represented by YSU. Scene number 10 of Maine imagery with 680 columns represented as MU. It is also an uncalibrated image with unsigned 12 bits per pixel information. The last image is scene number 1 of Hawaii represented

as HU, the number of columns equal to 614 and each pixel having information stored using 12 unsigned bits per pixels.

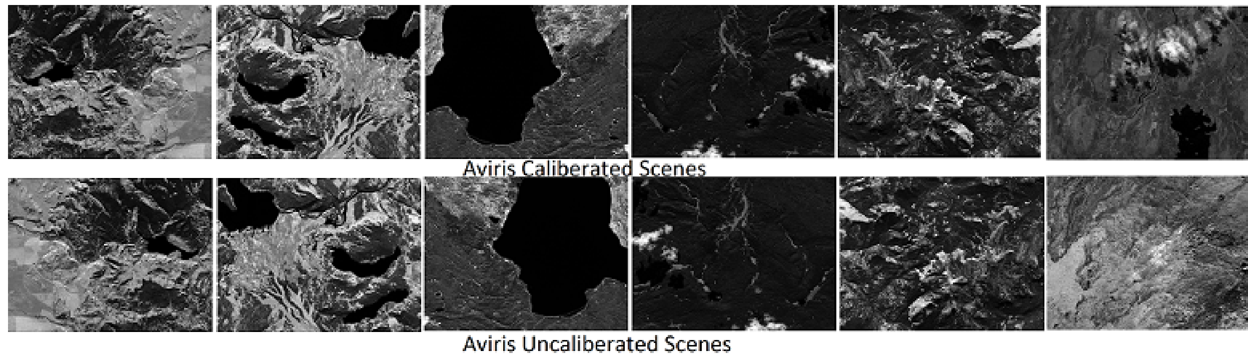


**Figure 4.5**: Grayscale image of CCSDS dataset used in the experiment

### Implementation environment and metrics

The experiment has been performed on the supercomputer "PARAM-SHIVAY", with the following configurations: Theoretical Peak Floating-point Performance is 837 TFLOPS, total number of CPU only compute nodes are 192, and total number of cores are 7680.

The implementation is done using Python 3.7 with numpy [169] and scipy library, and parallel programming is achieved using multiprocessing library pre-installed in python 3.7. The application utilizes only a limited number of nodes and memory from hardware as required, but any number of nodes and cores can be pulled anytime. In this section, the term 'processor' is used to represent the number of cores used in computation to remove the redundancy with the name 'core'.

Performance metrics used to evaluate the proposed algorithm are MAE (equation 4.4), MAPE (equation 4.5), and Prediction time as a function of a varying number of sub-processes that are executed in parallel. Relative speed-up and work optimality of each method was also calculated using equation 4.6 and 4.7 respectively.

$$MAE = \frac{1}{W*H*Z}\sum_{z=1}^{Z}\sum_{y=1}^{W}\sum_{x=1}^{H}(Absoulte(\tilde{S}(x,y,z)-S(x,y,z))) \qquad (4.4)$$

$$MAPE = 100 * \sum_{z=1}^{Z}\sum_{y=1}^{W}\sum_{x=1}^{H}(\frac{Absoulte(\tilde{S}(x,y,z) - S(x,y,z))}{S(x,y,z)}) \tag{4.5}$$

$$\text{Relative Speedup}=\frac{T_p\,(1)}{T_p\,(n)} \tag{4.6}$$

$$\text{Efficiency}\,(\eta)=\frac{T_p\,(1)}{T_p\,(n)\cdot n} \tag{4.7}$$

$T_p\,(1)$ = Time taken by parallel algorithm on 1 processor, $T_p\,(n)$ = Time taken by parallel algorithm on $n$ processors, $n$ = number of processors. $\tilde{S}\,(x,y,z)$ , $S(x,y,z)$ represent reconstructed pixel value and the original pixel value of HSI respectively, W represents width, H represents height, and Z represents number of bands/channels in HSI.

**Results**

This sub-section contains the numerical results of the experiment performed on proposed algorithms. The experiment was first carried out on all 12 images by executing the sequential RLS algorithm, followed by RLS-ALP, CRLS, and Fast-RLS-ALP sequential algorithms. The results of MAE (mean absolute error) and MAPE (mean absolute percentage error) were calculated for each algorithm separately. The sequential run-time of each algorithm was calculated by taking an average run-time of 10 consecutive executions.

The process was repeated by changing the number of bands from 4 to 100 in steps of 4 and changing the size of the context window for spatial decorrelation from $\{4, 12, 24\}$. The optimal prediction factor was found to be 28, and the context window size doesn't have much impact on MAE and MAPE.

Parallel algorithms were implemented with varying number of sub-processes and average run-time of 10 consecutive executions of each algorithm, for every sub-process was calculated. Table 4.2 contains the results of MAE and MAPE obtained for CCSDS images. It can be observed that the HU and MU dataset are the best performing while uncaliberated were the images giving maximum error in all the cases. The difference in

error among the sequential and parallel were incosiderable in almost all the algorithms. It contains the values for both parallel and sequential algorithms.

**Table 4.2**: Mean absolute error and mean absolute percentage error of CCSDS dataset

| Algorithm | RLS | | CRLS | | RLS-ALP | | Fast-RLS-ALP | | RLS-parallel | | CRLS-parallel | | RLS-ALP-parallel | | Fast-RLS-ALP-parallel | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Image | MAE | MAPE | MAE | MAPE | MAE | MAPE | MAE | MAPE | MAE | MAPE | MAE | MAPE | MAE | MAPE | MAE | MAPE |
| YSC-0 | 4.576 | 0.3754 | 3.856 | 0.3522 | 4.215 | 0.3627 | 4.213 | 0.3626 | 4.529 | 0.3752 | 3.854 | 0.3524 | 4.215 | 0.3621 | 4.214 | 0.3628 |
| YSC-3 | 4.511 | 0.3666 | 3.956 | 0.3501 | 4.111 | 0.3589 | 4.111 | 0.3589 | 4.623 | 0.3822 | 3.956 | 0.3503 | 4.113 | 0.3594 | 4.113 | 0.3594 |
| YSC-10 | 2.959 | 0.1863 | 2.468 | 0.1498 | 2.716 | 0.1662 | 2.716 | 0.1661 | 3.598 | 0.1866 | 2.446 | 0.1494 | 2.716 | 0.1652 | 2.725 | 0.1669 |
| YSC-11 | 3.659 | 0.1654 | 3.157 | 0.0794 | 3.384 | 0.1251 | 3.388 | 0.1251 | 3.649 | 0.1655 | 3.159 | 0.0799 | 3.389 | 0.1256 | 3.388 | 0.1251 |
| YSC-18 | 4.549 | 0.3258 | 3.869 | 0.3549 | 4.468 | 0.3179 | 4.467 | 0.3176 | 4.937 | 0.3118 | 3.958 | 0.3552 | 4.488 | 0.3178 | 4.479 | 0.3179 |
| Average | 4.051 | 0.2839 | 3.461 | 0.2573 | 3.779 | 0.2661 | 3.779 | 0.266 | 4.267 | 0.2843 | 3.475 | 0.2574 | 3.784 | 0.266 | 3.784 | 0.2664 |
| YSU-0 | 7.877 | 1.005 | 7.256 | 0.8125 | 7.684 | 0.8956 | 7.695 | 0.8958 | 7.857 | 1.0051 | 7.259 | 0.8125 | 7.684 | 0.8956 | 7.695 | 0.8957 |
| YSU-3 | 6.721 | 0.9554 | 6.165 | 0.7568 | 6.666 | 0.8995 | 6.262 | 0.8991 | 6.719 | 0.9555 | 6.159 | 0.7566 | 6.669 | 0.8999 | 6.668 | 0.8998 |
| YSU-10 | 5.816 | 0.7373 | 5.266 | 0.5967 | 5.528 | 0.678 | 5.527 | 0.6781 | 5.816 | 0.7372 | 5.268 | 0.5967 | 5.529 | 0.6782 | 5.521 | 0.6782 |
| YSU-11 | 6.252 | 0.8595 | 6.149 | 0.8494 | 6.591 | 0.9258 | 6.591 | 0.9258 | 6.282 | 0.8599 | 6.159 | 0.7582 | 6.592 | 0.9256 | 6.592 | 0.9258 |
| YSU-18 | 6.713 | 0.9543 | 5.891 | 0.7856 | 6.289 | 0.9248 | 6.289 | 0.9336 | 6.722 | 0.9466 | 5.895 | 0.7858 | 6.345 | 0.9256 | 6.289 | 0.9336 |
| Average | 6.676 | 0.9005 | 6.145 | 0.7602 | 6.552 | 0.8648 | 6.473 | 0.8664 | 6.679 | 0.9009 | 6.148 | 0.7419 | 6.654 | 0.8649 | 6.553 | 0.8666 |
| MU | 2.992 | 0.2745 | 2.156 | 0.1849 | 2.491 | 0.2248 | 2.491 | 0.2248 | 2.991 | 0.2745 | 2.159 | 0.1859 | 2.056 | 0.1246 | 2.061 | 0.2341 |
| HU | 2.599 | 0.1871 | 1.975 | 0.0948 | 2.056 | 0.1241 | 2.057 | 0.1241 | 2.589 | 0.1862 | 1.985 | 0.0845 | 2.165 | 0.1249 | 2.058 | 0.1242 |
| Average | 2.795 | 0.2308 | 2.065 | 0.1398 | 2.273 | 0.1744 | 2.274 | 0.1744 | 2.79 | 2.3035 | 2.072 | 0.1352 | 2.111 | 0.1248 | 2.059 | 0.1792 |

**Table 4.3**: Speedup, Efficiency and Run-time (in seconds) obtained on multiple processors

| Number of processors | RLS | | | CRLS | | | RLS-ALP | | | Fast-RLS-ALP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Speedup | Efficiency | Runtime | Speedup | Efficiency | Runtime | Speedup | Efficiency | Runtime | Speedup | Efficiency | Runtime |
| 7 | 6.2018 | 0.88597 | 518.1964 | 6.0161 | 0.85944 | 1727.585 | 5.6832 | 0.8119 | 564.141 | 5.4067 | 0.77238 | 182.7508 |
| 8 | 6.7739 | 0.8467 | 475.6165 | 6.9854 | 0.873175 | 1487.864 | 6.359 | 0.79485 | 504.1848 | 5.997 | 0.7496 | 163.758 |
| 10 | 8.2614 | 0.82614 | 389.1283 | 9.9817 | 0.99817 | 1041.238 | 7.668 | 0.7668 | 417.9484 | 8.573 | 0.8573 | 114.8055 |
| 14 | 11.3121 | 0.80801 | 284.5069 | 13.1872 | 0.9419 | 788.1371 | 10.843 | 0.7745 | 295.6128 | 9.125 | 0.65178 | 107.5958 |
| 16 | 11.8223 | 0.7389 | 271.9258 | 13.9504 | 0.8719 | 745.0196 | 11.0464 | 0.6904 | 290.1091 | 9.781 | 0.61131 | 100.2396 |
| 28 | 18.8516 | 0.6732 | 170.6736 | 19.6491 | 0.7017 | 528.9465 | 17.796 | 0.707 | 180.0807 | 16.083 | 0.5744 | 63.00263 |
| 32 | 19.0778 | 0.5961 | 168.5835 | 20.4856 | 0.6402 | 507.3477 | 17.669 | 0.55215 | 181.5239 | 18.1454 | 0.56704 | 54.1126 |
| 56 | 22.0086 | 0.39301 | 147.9195 | 28.9475 | 0.5169 | 359.0404 | 19.7148 | 0.35205 | 162.9486 | 17.9807 | 0.3211 | 54.98278 |

For RLS, CRLS, RLS-ALP algorithm, spatial decorrelation and spectral decorrelation were also calculated separately by different number of processors along with the original method. Figure 4.6 shows the results with the help of a bar chart to compare the time taken for spatial and spectral decorrelation. Number of processors = 1 in the

figure 4.6 represents the runtime of parallel algorithm on one processor.

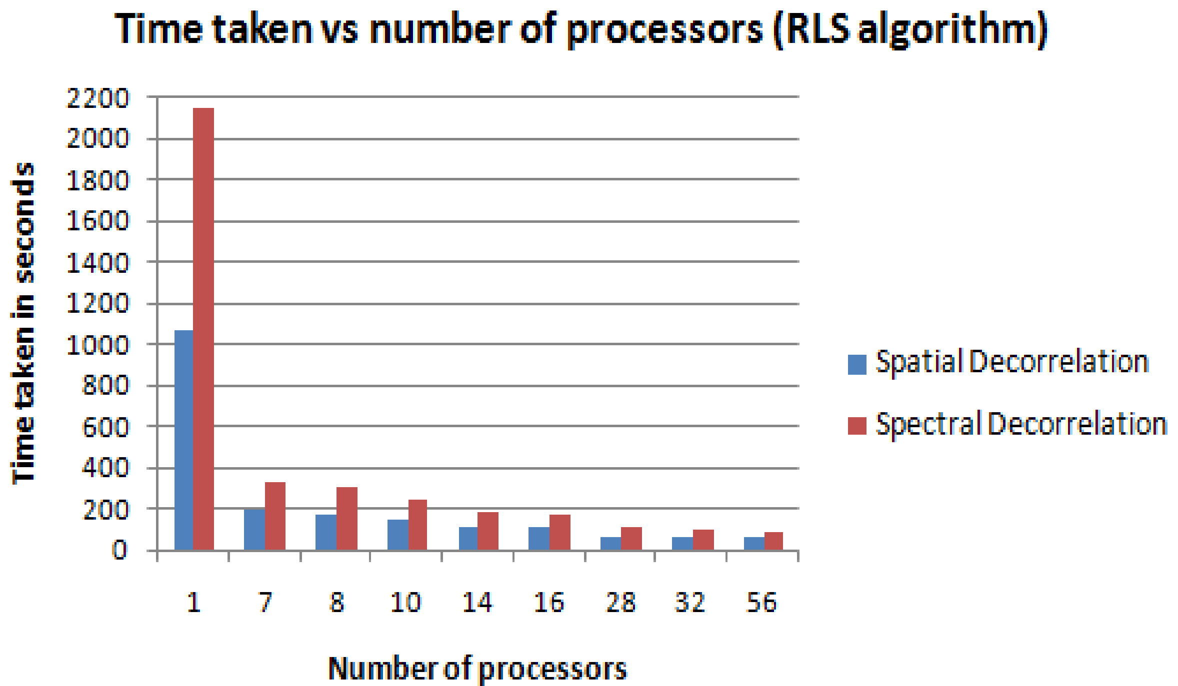


**Figure 4.6**: Runtime of spatial and spectral decorrelation separately

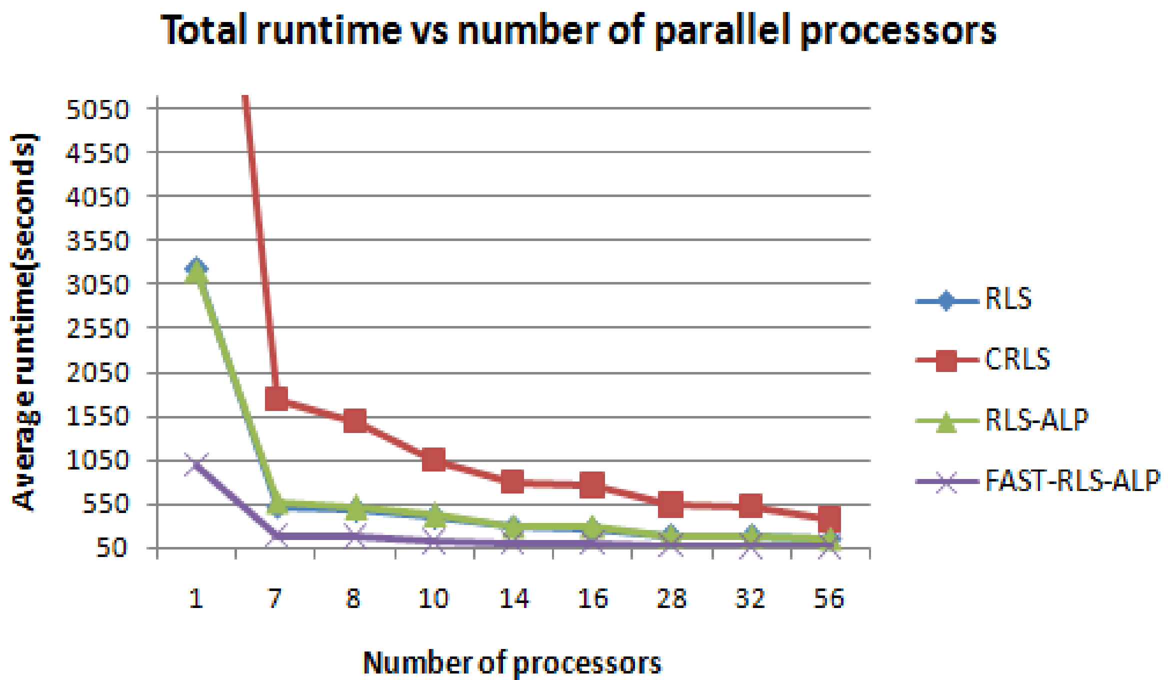


**Figure 4.7**: Total runtime vs number of processors

**Table 4.4**: Time taken (in seconds) to execute spatial decorrelation for CCSDS dataset by RLS algorithm

| Number of processors | 1 | 7 | 8 | 10 | 14 | 16 | 28 | 32 | 56 |
|---|---|---|---|---|---|---|---|---|---|
| YSC-0 | 1031.368 | 194.183 | 181.379 | 149.745 | 107.651 | 106.55 | 68.154 | 69.419 | 67.813 |
| YSC-3 | 1007.378 | 191.379 | 170.271 | 151.823 | 107.464 | 105.507 | 66.532 | 69.459 | 72.189 |
| YSC-10 | 1251.543 | 189.906 | 178.938 | 149.13 | 103.766 | 102.786 | 58.933 | 57.711 | 47.137 |
| YSC-11 | 1015.062 | 192.227 | 185.07 | 148.491 | 111.501 | 108.08 | 66.251 | 67.534 | 53.846 |
| YSC-18 | 1253.905 | 193.226 | 169.872 | 147.683 | 110.43 | 106.525 | 69.355 | 70.324 | 51.919 |
| YSU-0 | 991.658 | 190.454 | 168.561 | 150.08 | 104.193 | 105.799 | 65.561 | 67.255 | 52.935 |
| YSU-3 | 1153.28 | 191.131 | 173.382 | 149.74 | 105.319 | 105.702 | 63.483 | 57.892 | 47.833 |
| YSU-10 | 1136.578 | 194.429 | 177.875 | 148.774 | 105.312 | 105.909 | 68.202 | 70.993 | 67.316 |
| YSU-11 | 1054.316 | 196.858 | 180.079 | 158.328 | 113.98 | 110.394 | 69.763 | 62.244 | 48.6403 |
| YSU-18 | 994.232 | 185.355 | 171.161 | 147.254 | 105.705 | 101.738 | 67.254 | 68.282 | 67.339 |
| MU | 1026.662 | 193.8798 | 189.445 | 154.297 | 111.221 | 108.442 | 67.974 | 70.823 | 69.393 |
| HU | 926.856 | 172.478 | 161.311 | 137.051 | 97.5571 | 96.011 | 58.2189 | 52.824 | 42.8048 |

**Table 4.5**: Time taken (in seconds) to execute spectral decorrelation for CCSDS dataset by RLS algorithm

| Number of processors | 1 | 7 | 8 | 10 | 14 | 16 | 28 | 32 | 56 |
|---|---|---|---|---|---|---|---|---|---|
| YSC-0 | 2153.269 | 334.636 | 313.694 | 245.728 | 178.638 | 164.601 | 100.6 | 109.3696 | 99.601 |
| YSC-3 | 2198.837 | 320.991 | 269.313 | 246.795 | 177.014 | 172.305 | 95.436 | 96.204 | 83.669 |
| YSC-10 | 2188.978 | 337.872 | 340.291 | 236.73 | 180.283 | 165.759 | 107.145 | 105.785 | 78.029 |
| YSC-11 | 2093.89 | 329.8 | 292.09 | 245.651 | 183.368 | 170.103 | 111.059 | 98.21 | 83.04 |
| YSC-18 | 2280.162 | 348.193 | 300.953 | 242.825 | 179.882 | 174.634 | 109.623 | 108.111 | 89.0045 |
| YSU-0 | 2168.597 | 310.328 | 318.544 | 231.827 | 166.28 | 162.084 | 98.985 | 99.56 | 87.429 |
| YSU-3 | 2057.414 | 314.999 | 281.436 | 231.624 | 173.653 | 161.8836 | 103.644 | 112.633 | 90.853 |
| YSU-10 | 2183.951 | 342.062 | 279.009 | 245.469 | 172.689 | 178.707 | 110.412 | 102.504 | 93.265 |
| YSU-11 | 2158.951 | 329.174 | 319.751 | 246.988 | 202.914 | 167.515 | 112.192 | 104.668 | 95.493 |
| YSU-18 | 2119.385 | 332.718 | 285.549 | 240.301 | 175.486 | 169.48 | 100.3408 | 100.9603 | 89.335 |
| MU | 2131.145 | 331.376 | 329.675 | 240.619 | 178.441 | 164.221 | 110.333 | 102.667 | 115.59 |
| HU | 1990.683 | 300.702 | 269.749 | 222.587 | 161.336 | 148.3745 | 98.6325 | 97.5702 | 80.5601 |

The effect of the use of HPC can be visualized in Figure 4.7, where the total run-time of different algorithms of prediction-based compression is presented in the form

of a graph. It shows the gradual decrease in time when the algorithm is computed in a parallel environment as compared to the sequential one. Two lines representing RLS and RLS-ALP overlap since both have the same computation except the number of previous bands used to predict the values of pixels of a band. However, they differ in terms of numerical values slightly, which can be seen in Table 4.3, it also contains the values of speedup by the different number of processors. Graphically this parameter is represented in Figure 4.8 for speedups obtained by different algorithms, the minimum value obtained is 5.4067 and maximum is 28.9475. Table 4.4 contains time taken to remove the spatial correlation from an image sequentially and on different number of processors. Similarly, Table 4.5 contains the run-time of spatial decorrelation by the RLS algorithm.

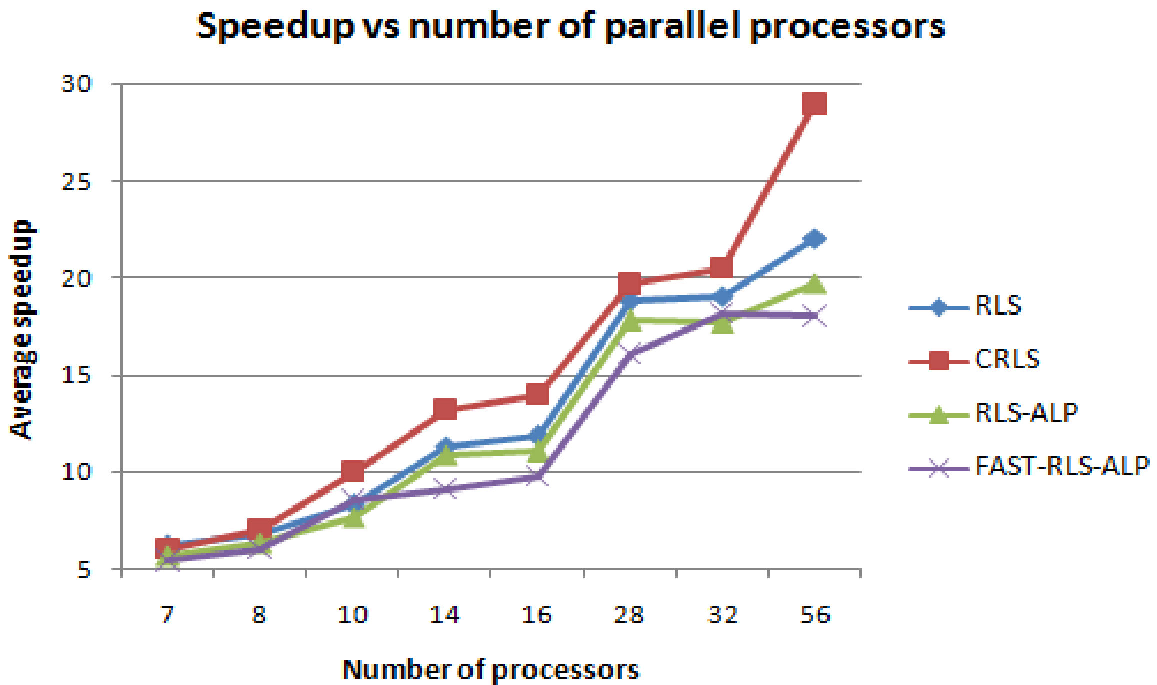


**Figure 4.8**: Speedup vs number of processors

**Work Optimal Solution:** In the HPC environment, a theoretical concept of work optimality is used to check the performance of an algorithm based on number of processing elements consumed. It can be calculated using equation 4.7, for each number of

sub-processes. The concept is also known as efficiency. Figure 4.9 shows the efficiency of proposed parallel algorithms for the different number of processors.

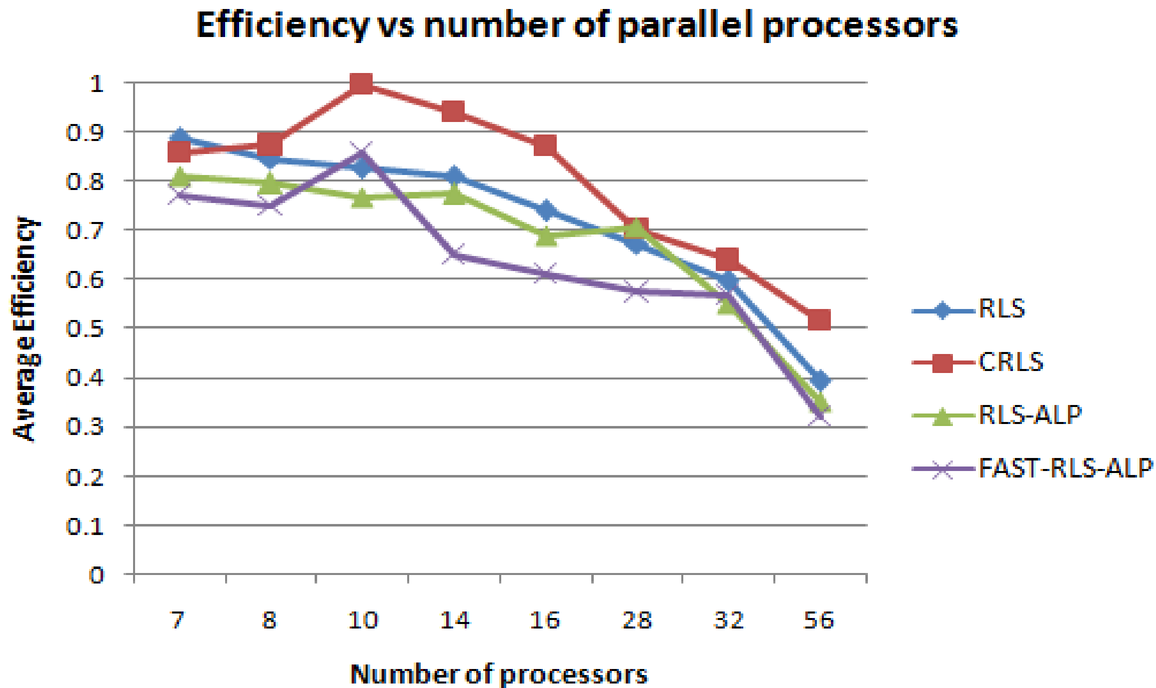


**Figure 4.9**: Efficiency vs number of processors

It shows the disadvantages of increasing the number of processors to get good speedup as most of the time is wasted as communication overhead. In this experiment, the optimal number of processors is obtained as 10, but if demand is to reduce the time of compression and decompression phase, work optimality has to be sacrificed.

In this experiment, four prediction based HSI compression algorithms are executed in HPC environment. A-CRLS method gives optimum compression results as compared to RLS and RLS-ALP with a drawback of high space and execution time. It can be visualized from the results that the best speedup and efficiency are obtained when A-CRLS is executed using parallel programming paradigms. However, it is noticeably worse in execution time, due to the inherent characteristics of the sequential algorithm (consumes 382% more time than CRLS). We have also executed spatial and spectral decorrelation separately by using a different number of processors. Results obtained in

this process represent almost similar behavior for spatial and spectral decorrelation by using the concept of data parallelism.

## 4.4 Conclusion

Parallel programming is used to reduce the run-time of RLS based compression algorithms. In this chapter, we have proposed a method to reduce the complex structure of prediction - based compression algorithms by using HPC architecture. The very first step is to remove the dependency within the program, which restricts the use of parallel programming. Then, a multi-processing model is used to reduce the run time of prediction algorithms. The program was executed on multi-node system by changing the number of processors in multiple iterations. RLS algorithm is modified to execute spatial and spectral decorrelation separately in different functions. Effect of increasing number of processing elements on execution time of proposed parallel algorithm is discussed in terms of speedup and work optimality.

This method of parallelization can be used in real-time prediction and compression of more complex data. The process can be more optimized by using the concept of hybrid programming, which includes shared memory and distributed memory parallelism together. Moreover, the method can be used to parallelize the compression of medical images used to detect the tumor, cancer using hyperspectral imaging.