

A Parallel Implementation of an Image Processing Algorithm

Jared O'Connell and Peter Caccetta

November 25, 2005

CSIRO Mathematics & Information Sciences

Private Bag 5, Wembley WA 6913

Ph: (08) 9333 6106

Email: jared.oconnell@csiro.au, peter.caccetta@csiro.au

Abstract

This paper describes the parallel implementation of an algorithm used for time-series classification of remotely sensed image data. Such classifications form the basis of many operational programs aimed at monitoring land use and cover change, and monitoring programs typically comprise data of the order of hundreds of megabytes to terabytes. The use of time-series models for classification has led to great increases in the accuracy of land-cover change estimates, at the cost of increased computation, being of the order of a CPU year for larger monitoring programs. In this paper we explore the use of clusters of computers for reducing computation times. We describe our port using the MPI standard, and give timing performance results on a homogeneous and a heterogeneous cluster, being examples of dedicated and opportunistic clusters respectively. Results for both clusters show good CPU efficiency leading to useful processing time reductions. We describe issues encountered while doing the port and subtle algorithm design issues associated with the two different clusters.

Keywords: *parallel image processing, land-cover change*

1 Introduction

In Australia, remotely sensed Landsat TM data is routinely used for mapping and monitoring the change in extent of woody perennial vegetation. Time-series remotely sensed satellite imagery and ground information are used to form multi-temporal classifications of presence/absence of woody cover. Current operational monitoring systems typically include:

- registration and spectral calibration of time-series Landsat data to a common reference
- processing of the calibrated data to remove corrupted data, which include; dropouts, data affected by fire, smoke and cloud
- stratification of the data into ‘zones’, where land-cover types within a zone have similar spectral properties;
- analysis of ground and spectral data to determine the appropriate single date classifier and its parameters.
- validation of the results to quantify the accuracy of the results

Typically the information derived has greater value as the timeliness and accuracy improves, being useful for an increasing array of natural resource management issues. Greater classification accuracies may be obtained by specifying a joint temporal/spatial model for multi-temporal classification. This increased accuracy comes at the cost of increased computation, which may become significant for large monitoring systems.

Here we consider parallel implementations of the spatial/temporal algorithms described by (Caccetta 1997), (Kiiveri & Caccetta 1998) and employed in the Australian Greenhouse Office (AGO) Land Cover Change (LCC) project (Furby 2002). The project estimates forest change for the continent from 1972 to the present, transforming terabytes of raw data into forest presence/absence images. The calculations take of the order of 1 CPU year to run so clusters of computers offer an attractive approach to reducing the physical

time to process the data.

In section 2 we describe the data and models. Section 3 discusses the existing serial implementation of the models, issues for parallel implementation on clusters and the parallel implementation we derived. In section 4 we describe two different clusters on which we test the performance of the parallel algorithm with results described in section 5. The first cluster we call an ad-hoc cluster as it was formed from our workplace desktops comprised of various vintages of Intel CPUs running the Windows operating system. The second cluster is a dedicated cluster operated by the CSIRO High Performance Computing Centre, and comprised of Xeon CPUs running the Linux operating system. We comment on lessons learnt in section 6.

2 Methodology

2.1 Data

Data in the LCC project involves satellite imagery covering a period of over thirty years, starting in 1972. Images come from three different sensors, Landsat MSS, Landsat TM and Landsat ETM⁺. Landsat MSS contains four spectral bands while TM and ETM⁺ each have six. Australia is divided into 37 1:1000000 map-sheets and an operator will typically work on one of these map-sheets at a time. There are several data preprocessing steps including rectification, calibration and mosaicing which we assume have been performed. This processing makes data spatially, spectrally and temporally comparable. Those interested may refer to (Furby 2002). The full time-series (13 dates at present) is of the order 25 to 30 GB per map-sheet or over 1 TB for the continent.

2.2 Preliminary classification/Likelihood calculation

After preprocessing, the data is classified to form a likelihood via thresholding of two or three spectral index. An index is a linear combination of different spectral bands. An image must be stratified by both scene date and zone (a zone typically covers a certain

soil/vegetation association or other geographical feature) with different indices used for different zones and different thresholds used for scene dates within zones. When mosaiced, these zones produce a mapsheet sized image for each time step with a probability of forest presence/absence assigned to each pixel.

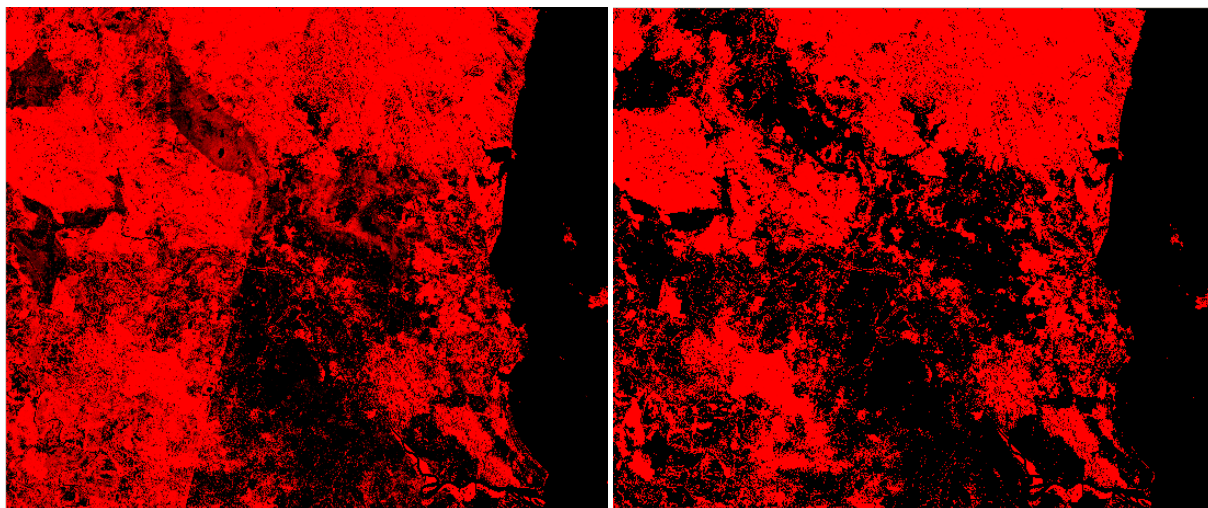


Figure 1: **Left panel:** A classified image, pixels with high intensity represent probabilities at or close to 1. Note the visible scene boundary, this is not a believable pattern of vegetation. **Right panel:** Spatial/temporal model output, the scene boundary has been corrected.

These probabilities represent the likelihood that an area is forest (Figure 1, left panel). The probability images contain two ‘bands’, $P(\text{Woody})$ and $P(\text{Non-woody})$ ¹, and hence are $\frac{1}{2}$ or $\frac{1}{3}$ the size of the original imagery. So a full time-series of classified images will be approximately 8 to 10 GB per 1:1000000 mapsheet or over 300 GB for the continent. These probability images contain omission errors (for example true forest not identified) and commission errors (for example non-forest areas identified as forest). These errors exist for a variety of reasons including poor or missing data (for example data with smoke or haze), limited spectral separation of some land-covers, and operator and classifier limitations. In extreme cases one can see obvious inconsistencies between scene or zone boundaries as is the case in Figure 1, left panel. The use of spatial/temporal models can significantly reduce many of these errors (Figure 1, right panel).

¹Here we note that we could store only one of these but for practical reasons we store the probabilities for each class.

2.3 Spatial/temporal models

In this section we describe the spatial/temporal models used for improving classification and comment on some of the possibilities for parallel computation. We first define some notation. For pixel $k = 1 \dots m$ and time slice $i = 1 \dots n$ we write y_{ik} , y'_{ik} , l_{ik} , l'_{ik} and z_{ik} to represent Landsat coverage, index image, initial interpretation, ‘true’ class label and stratification zone respectively. We use the notation $r(k)$ to denote the 8 pixels adjacent to k . The joint model for the observed and unobserved images can be written as

$$p(y'_{1k} \dots y'_{nk}, l'_{1k} \dots l'_{nk}, l_{1k} \dots l_{nk}, l'_{1r(k)} \dots l'_{nr(k)}, z_{1k} \dots z_{nk}) = \prod_{i=1 \dots n} p(y'_{ik} | z_{ik}, l_{ik}) p(l_{ik} | z_{ik}, l'_{ik}) p(l'_{ik} | l'_{i-1,k}, l'_{ir(k)}) p(z_{ik})$$

where $p(l'_{1k} | l'_{0k}, l'_{ir(k)})$ is defined as $p(l'_{1k} | l'_{ir(k)})$. Note that for operational projects it is convenient to perform calculations involving y'_{ik} , z_{ik} , l_{ik} in a separate process as described in section 2.2

The ‘true’ class labels are obtained from the joint distribution of $l'_{1k} \dots l'_{nk}$ given the observable images by a cyclic ascent algorithm as in (Caccetta, 1997 pp 187-192), (Kiiveri & Caccetta 1998). We will refer to this algorithm as CK in the following. Central to CK is a means to perform efficient computations for the above distribution. In the serial case of CK the algorithm described by (Lauritzen & Spiegelhalter 1988) is used and is denoted as LS in the following.

Useful properties of this approach include:

- propagation of uncertainties in inputs and calculation of uncertainties in outputs
- production of hard and soft maps (class label and probability)
- handling of missing data by using all available information to make predictions
- existence of well-developed statistical tools for parameter estimation.

Here we point out the three properties of the algorithm which are of interest for parallel computations.

1. the algorithm is iterative, making several passes of the data to cluster (time-series) pixel labels into homogeneous regions.
2. at each iteration, each pixel has a relatively expensive computation performed (which in the serial version makes use of the LS algorithm). This computation combines time-series information with neighbouring pixel information to produce an updated pixel label.
3. computations for each pixel are independent of all other pixels given the neighbouring pixel labels from the previous iteration.

Essentially, this means there is an expensive computation that has to be performed for each pixel a number of times.

There are two obvious ways to introduce parallel computations and hence reduce computational time. The first approach would be to parallelize step two: that is parallelize LS and have the cluster process each pixel in succession. (Madsen & Jensen 1999) provide a review of parallel algorithms suitable for this purpose. The second approach, and the one adopted here, is to exploit point 3 above and have the LS computation for each pixel processed by a different CPU.

3 Implementation

The original LS implementation was written in C (Caccetta 1997) and has been ported to various platforms throughout its lifetime. The code was modified to exploit parallelism using the Message Passing Interface (MPI) library (Snir, Otto, Huss-Lederman, Walker, & Dongarra 1995). MPI is a commonly used standard in parallel computing and many implementations are available. Several MPI implementations were used in this project and these are detailed in section 5.

3.1 Serial Algorithm

The pseudo code for the critical section of the serial implementation is shown below. The first loop controls the number of iterations, the second loop iterates through the number of lines in the image and the inner loop iterates through each pixel on line l . It is important to note that to calculate probabilities (and hence class labels) using a spatial dependence model one needs the class labels of the previous, current and next line in addition to the input probability for the current line. That is, to produce estimates for line l one also needs class labels from lines $l - 1$ and $l + 1$.

Pseudo code for the serial algorithm follows

```
1: procedure SERIAL CK
2:   for ( $i$  in  $No\_Iterations$ ) do
3:     for ( $l$  in  $No\_Lines$ ) do
4:       Read_Input_Probabilities( $l$ )
5:       if ( $i > 1$ ) then ▷ Spatial dependence after first iteration
6:         Read_Class_Labels( $l$ )
7:       end if
8:       for ( $j$  in  $width$ ) do
9:         LS_Processing( $j$ )
10:      end for
11:      if ( $i < LAST\_ITERATION$ ) then
12:        Buffer_Class_Labels( $l$ ) ▷ Class labels for neighbourhood processing
13:      else
14:        Write_Output_to_HDD( $l$ )
15:      end if
16:    end for
17:  end for
18: end procedure
```

Note that the first iteration involves spatially independent calculations. Class labels (not soft probabilities) from this and following iterations are buffered and then used for neighbourhood processing on successive iterations. As mentioned previously, we are dealing with large data sets, a typical size for one of these class label buffers may be $13 \times 26800 \times 18800$ bytes (6.5 GB). This is not yet practical to store in RAM and some sort of compression is required. Fortunately, this data can be quickly and efficiently compressed using simple run length encoding (RLE) (Golomb 1966). The class labels typically deflate to 5% of their original size with RLE compression.

3.2 Parallel Algorithm

There is obvious potential for distributing the algorithm described in the previous section between multiple CPU's. One can simply distribute each line of the input probability images (along with respective neighbouring class labels) to a number of slave nodes with I/O performed by one master node. Slave nodes then perform the necessary LS processing tasks and send the output back to the master. The output being soft probabilities for pixels and their corresponding class labels. Since only class labels are required for neighbourhood processing the soft probabilities need only be sent back to the master on the last iteration.

Pseudo code for such a technique follows

```
1: procedure PARALLEL CK - MASTER CODE
2:   for (i in No_Iterations) do
3:     for (s in min[No_Nodes, No_Lines]) do   ▷ Send a line to each slave initially
4:       l = s                                     ▷ l = current to line to send out
5:       Read_Input_Probabilities(l)
6:       Send_Iteration(s)
7:       Send_Probabilities(s)
8:       if (i > 1) then   ▷ Neighbourhood calculations after first iteration
9:         Read_Class_Labels(l)   ▷ Read class labels into a buffer
10:        Send_Class_Labels(s)
11:      end if
12:    end for
13:    No_Received = 0
14:    while (No_Received < Total_No_Lines) do
15:      s = Recv_Class_Labels()   ▷ Receive class labels from any slave
16:      if (i = LAST_ITERATION) then
17:        Recv_Probabilities(s)   ▷ Soft probabilities required for final output
18:      end if
19:      Read_Input_Probabilities(l)
20:      Send_Iteration(s)
21:      Send_Probabilities(s)
22:      if (i > 1) then
23:        Read_Class_Labels(l)   ▷ Read class labels into a buffer
24:        Send_Class_Labels(s)
25:      end if
26:      if (i = LAST_ITERATION) then
27:        Write_Output()           ▷ Flushes the buffers if possible
28:      end if
29:    end while
30:  end for
31:  Broadcast_Finish_Message()
```

32: **end procedure**

The master first sends a line to each slave for processing (for loop, line 3). The *min* term is included purely for rigor, we are yet to see a practical example where slaves outnumber image lines. There is no requirement for processed lines to be received in order by the master, they may simply be buffered (this prevents a slow slave delaying the whole cluster). The *Write_Output()* function writes as much to disk as possible and then frees the buffers.

The slave code is as follows

```
33: procedure PARALLEL CK - SLAVE CODE
2:   repeat
3:     i = Receive_Iteration()
4:     Recv_Probabilities()
5:     if (i > 1) then                                     ▷ Spatial dependence after first iteration
6:       Recv_Class_Labels()
7:     end if
8:     for (i in width) do
9:       LS_Processing(i)
10:    end for
11:    Send_Class_Labels(master)
12:    if (i = LAST_ITERATION) then
13:      Send_Probabilities(master)   ▷ Soft probabilities required for final output
14:    end if
15:  until Finished message received
16: end procedure
```

An inherent limitation to the number of CPUs that can be effectively used with an algorithm of this nature is the rate at which the master node can send and receive data. We do not appear to have hit this limit yet but it may become an issue when the cluster size increases. The results in the section 5 demonstrate that, despite its simplicity, this algorithm makes efficient use of computing resources and has scaled well in the current environment.

3.3 Notes on the Evolution of the Parallel Code

This code underwent several revisions before we arrived at the implementation described in the previous section. The initial implementation was largely a proof of concept, it was implemented as quickly as possible with the knowledge that there were inherent inefficiencies in its design. For example, early implementations made no allowance for processed lines arriving at the master out of order on the last iteration (lines are written to the disk sequentially). This meant that faster slaves were forced to wait for slower slaves. Early implementations also made poor use of compression, with processed lines originally sent back to the master and then were compressed and buffered by the master (subsequently we have the slaves doing the compressing). Even with the above limitations we observed significant gains in speed. Some very informal testing found we had achieved close to linear scalability and around 40% efficiency. This demonstrated that a useful speed increase was possible and so a more sophisticated rewrite was in order.

The first attempt at parallel code was essentially a hack of the original I/O functions in the serial software. Besides improving code clarity, the new design involved writing custom, optimised I/O functions. A primitive sliding window algorithm was used to circumvent the problem of unordered lines. Lines that arrived out of order were buffered until the required preceding lines arrived. This superior implementation was the subject of the more formal testing described in in section 5.

Also considered, but not implemented, was the compression of input probabilities sent by the master to slaves. This could be an advantage for widely geographically distributed ad-hoc clusters or other clusters where lack of high-speed communication may be a limitation.

4 Hardware

4.1 Ad-Hoc Cluster

Initial development was done on an ad-hoc cluster of 13 office Wintel machines. Machines were of varying age and hence varying speed. All machines were Pentium 4's between 1.6Ghz and 3.6Ghz and of single, dual and quad CPU varieties connected via 100mbit ethernet. Each node had varying amounts of memory however this was not a concern considering the small amount of data a slave process keeps in buffers at any one time. This heterogeneous environment was far from ideal as machines were not dedicated so there was no guarantee of 100% of a machine's cycles. The operating system and hardware were in no way optimised for parallel computation. In spite of these facts, the ad-hoc cluster provided a useful and immediate test-bed and yielded a significant decrease in processing time. The MPICH (Gropp, Lusk, Doss & Skjellum 1996) implementation was used for this cluster, it is one of the few free MPI implementations available for Windows. Code was compiled using the Microsoft Visual C++ 6 compiler.

4.2 Dedicated Cluster

After initial development, a dedicated and more powerful cluster was put to use. The CSIRO High Performance Computing Centre provides a commodity cluster for general computation comprised of 34 nodes with dual 3.2 GHz Xeon processors connected via Gigabit ethernet. This computer ran the Linux (Redhat 9) operating system and one could make use of either the MPICH2 or LAM implementations of MPI. The Intel C compiler was also available in this environment and was found to produce faster binaries than the GNU compiler. The results in the next section were produced using MPICH2 which seemed to perform slightly better than LAM in some brief initial testing, this may be purely due to the compiler configurations. While the ad-hoc cluster proved useful, this environment clearly provided more power and its homogenous nature allowed us to better measure the scalability and efficiency of our implementation.

5 Performance

To measure the decrease in time and scalability of the code a small data set was repeatedly analysed using different configurations on both the ad-hoc and dedicated clusters. A data set containing probability inputs from eleven years with images 19470 pixels wide and 1000 pixels high was used. The number of time steps and width of the image are fairly typical in applications while the height was shortened to hasten the testing process. It should be clear from the algorithm that processing time will be linear with respect to height (this was informally verified).

5.1 Ad-hoc Cluster Performance

It was impossible to give any genuine measures of scalability or efficiency of the code when running on our ad-hoc cluster due to its heterogenous nature. There were only three sets of two machines that had the same specifications. We can however discuss some time measurements less formally.

The results in Table 1 show a reasonable decrease in time given the less than ideal conditions. To be conservative, we compare all times to a serial run using the fastest available machine (Pentium 4 3.6Ghz). Running a cluster with five slave CPU's of similar speed we see a decrease in time by a factor of 3.4. This suggests reasonable efficiency, even if this was a homogenous set of CPUs this would indicate an efficiency of at least 67%. A run on the full cluster of 13 CPUs saw a speedup of 6.7 yielding a decrease in time from days to hours, making this a useful operational tool. Considering three of these CPUs were only 1.6Ghz this suggests good efficiency.

Slave CPUs	Serial	5 ⁺	6 [*]	13
Time (s)	15204	4480	5408	2250
Serial Time/Time	1	3.39	2.81	6.76

Table 1: Performance results for our ad-hoc cluster.⁺This trial used 2 3.2 Ghz machines and 3 3.6 Ghz machines.^{*}This trial had a master and slave process running on 1 CPU. This demonstrates the need for a dedicated master.

The design of the algorithm aimed to minimise the work done by the master process, it should be doing nothing but I/O. With this in mind, it was thought that a CPU may be able to act as both a slave and master. It was quickly found this was not the case as the results in Table 1 show, a cluster with a dedicated master does significantly better than one without. It is important to note that in these tables only the number of slave CPUs is specified.

5.2 Dedicated Cluster Performance

Not surprisingly, the dedicated cluster yielded far larger speed increases than our ad-hoc cluster. The same 1000 line data set was used in testing the algorithm on clusters of sizes varying between two and fifty CPUs (remembering one of these CPUs runs the master process).

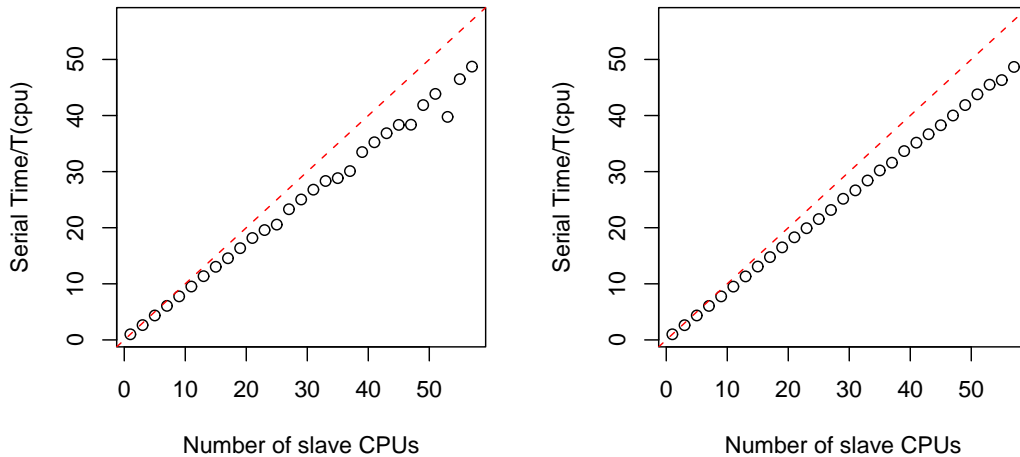


Figure 2: Speedup against number of CPUs. Left panel: Total time taken against number of slave CPUs. Right panel: Time for second iteration against number of slave CPUs. Some minor I/O issues need to be resolved.

While the dedicated cluster generally gave good performance, we noticed some times did not fit the general trend of the speedup factor (Figure 2, left panel). It was eventually found that this could be attributed to I/O issues. Occasionally, reading the initial image lines from the HDD would take a relatively long time (sometimes 20 s longer than usual).

This minor problem could be resolved by buffering the entire image using RLE, leading to a large reduction in I/O requirements. If we look at the time taken for the second iteration we see a much smoother trend as there is no delay for initial HDD seek time (Figure 2, right panel).

Slave CPUs	Serial	9	19	29	39	49	79
Time (s)	6042	778.00	369	241	180	144	87.88
Serial Time/Time	1	7.76	16.35	25.04	33.49	41.86	68.8
Efficiency	100%	86%	86%	86%	86%	85%	87%

Table 2: Performance results for the CSIRO HPC dedicated cluster.

Ignoring these occasional reductions in efficiency caused by slow I/O (which could also happen when running serial code) we see close to linear scalability (Figure 2, Left panel) and good efficiency (Table 2). A 9 CPU cluster produces a speedup of 7.76 (86.2% efficiency) and a 49 CPU cluster yields a reduction in time by a factor of 41.86 (85.43% efficiency). So reductions in efficiency as the number of CPUs increase are small, if any. It does not appear that computation is being bound by the ability of the master to send and receive data. This is also indicated by the profiling results shown in Figure 3 where we see processing time is dominated by the LS computation.

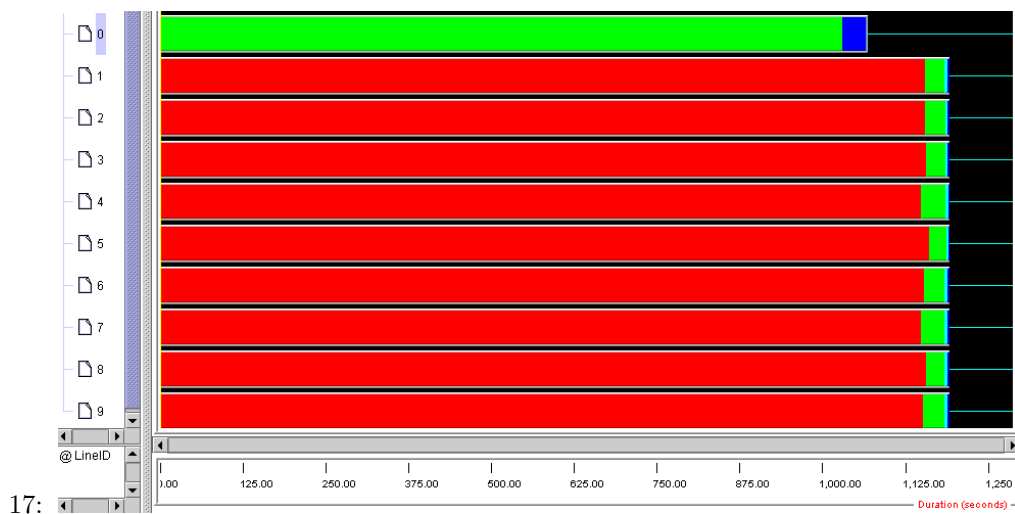


Figure 3: Histogram of profiling results for a 10 node run on the dedicated cluster. Node 0 is the master. LS processing is red, receiving state is green and sending state is blue.

6 Conclusions

This paper described the porting of a heavily used image processing implementation to run in parallel using the MPI standard. Performance is measured on both ad-hoc and dedicated clusters. The parallel algorithm achieves good efficiency and scalability. Some implementation (particularly I/O) issues remain but from an operational standpoint these are not of any great concern. That aside, this algorithm works well within the environments it is currently being used.

The reduction in computation time has several practical benefits. Firstly, turn around time per map-sheet has been reduced from days to hours for this part of analysis. This is beneficial as it is now less of an ordeal to have operators re-run a calculation if problems are observed (map revision). Secondly, it raises the possibility of increasing the accuracy of land-cover change by; making the use of denser time-series possible (more image dates), having more complicated spatial/temporal models (eg. two or three time-step dependence), performing a larger number of iterations. This implementation has already proved itself as an extremely useful operational tool.

References

- Caccetta, P. A. (1997). *Remote Sensing, GIS and Bayesian Knowledge-based Methods for Monitoring Land Condition*, PhD thesis, School of Computing, Curtin University of Technology.
- Furby, S. L. (2002). Land cover change: Specification for remote sensing analysis, *Technical Report 9*, Australian Greenhouse Office.
- Golomb, S. (1966). Run-length encodings (corresp.), *IEEE Transactions on Information Theory* **12**: 399–401.
- Gropp, W., Lusk, E., Doss, N. & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard, *Parallel Computing* **22**(6): 789–828.
- Kiiveri, H. T. & Caccetta, P. A. (1998). Image fusion with conditional probability networks for monitoring salinisation of farmland, *Digital Signal Processing* **8**(4): 225–230.
- Lauritzen, S. L. & Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems, *J. Roy. Statist. Soc. Ser. B* **50**(2): 157–224. With discussion.
- Madsen, A. & Jensen, F. (1999). Parallelization of inference in bayesian networks, *Technical Report R-99-5002*, Department of Computer Science, Aalborg University.
- Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., & Dongarra, J. (1995). *MPI: The Complete Reference*, MIT Press.