Message-Passing Programming: 2-D Image Recomposition Coursework

Jack Tyler: 27513556

27 May, 2019

In the early days of computing, computational power progressed at an alarming rate; constant field scaling meant that transistors could be fabricated smaller and smaller, allowing them to use less power, and increasing their clock speed. However, transistor manufacture has hit a physical limit for scaling (McFarland and Flynn, 1995), and increasing the clock speed any further would require highly capable cooling solutions impractical for use. As a result, processor companies have had to find other methods to speed up computer processors to keep computational power, and Moore's Law, advancing. Perhaps most prominent is the introduction of distributed parallelism, where multiple processor units, which otherwise operate asynchronously and independently, may communicate while performing some task.

This type of parallelism is one of the most common implementations currently used in high-performance computing to decrease program execution time. However, care must be taken as to data synchronisation occurs between processors: since distributed parallelism does not share memory, data must be explicitly passed between any workers in a pool.

This report investigates the performance of a two-dimensional image re-composition program; in particular, it studies the strong scaling for the program, and analyses several methods of handling file operations. This work will also identify many of the limitations and shortcomings in both the parallelisation of the algorithm, and in the analysis method. Knowledge of the Message-Passing Interface (MPI) standard, parallel computing nomenclature and the implementation of the one-dimensional cartesian topology is assumed in this report.

Introduction

Edge detection methods, such as the Sobel-Feldmann operator (Sobel, 2014) or the Canny method (Canny, 1986), are widely used in the fields of image segmentation and computer vision, and many algorithms can often be performed in only several iterations. However, the inverse operation – of recreating an image given its edges – is a more complex process, and often requires a large number of iterations. With images representing a rectangular compute domain that is readily divisible, image re-composition is a problem that lends itself well to being parallelised.

In the case of this work, let $L\left(u,\ v\right)$ be a local co-ordinate system attached to every pixel in an image (noting that a pixel is some value between 0–255, where 0 represents a fully black pixel and 255 fully white) with v defined as parallel to the gradient direction. If we then assume that the image has been pre-smoothed and can be associated with some scale-space operator $L\left(x,\ y\right)$ then we may define an edge as the zero-crossing of the second derivative of L. That is, that we mark a pixel as an edge if the following condition holds:

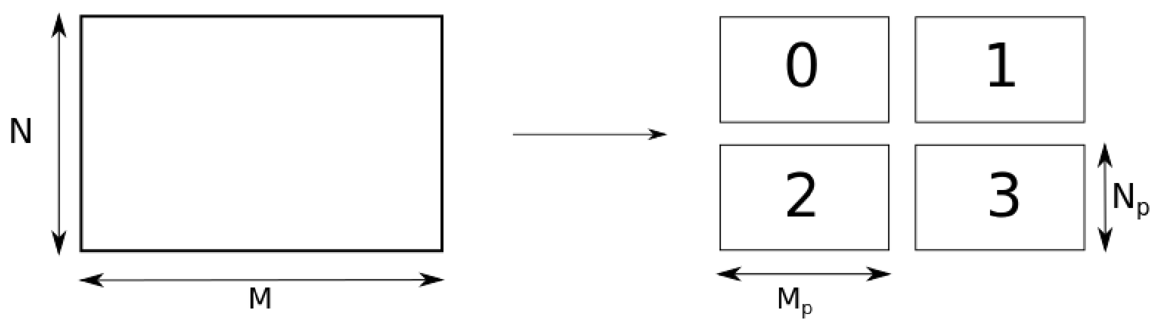


Figure 1: Two-dimensional domain decomposition used in this work. For some M-by-N pixel image, each processor (in this case 4) can be compute an M_p-by-N_p chunk of the image.

$$\frac{\partial^2}{\partial v^2} L = 0$$

Similarly, we may perform the inverse operation: knowing the second derivative of some scale-space operator allows us to determine the original value of the pixel; for example, this work uses a Jacobi iterator to approximate the original value of the pixel from the second derivative.

$$L(x,\ y) = \frac{1}{4}\left(L(x+1,\ y) + L(x,\ y+1) + L(x-1,\ y) + L(x,\ y-1) - L(x,\ y)_{xx}\right)$$

Parallelising the Algorithm

Since this is a continuous problem, it can be computationally intensive for large images. The use of MPI and multiple, distributed processors, can therefore be used to decrease program run-time.

Consider Figure 1: for some image that has size M-by-N pixels, we may use a domain decomposition technique to split the image in smaller chunks, and each processor may then compute only its local chunk of the whole image. The MPI standard defines intrinsic routines to perform this Cartesian domain decomposition, and as such, the implementation of this decomposition is trivial. However, handling the initial dis- and re-assembly of the image requires careful implementation.

A description of the parallel algorithms' behaviour follows:

P instances of the program are launched via the `mpirun` or `aprun` directives, and the MPI communication routines are spawned using the `MPI_Init` subroutine. For portability, all of the MPI routines are wrapped in a library which mimics the serial subroutine calls, and abstracts the parallel modifications from the user.

Also spawned along with the MPI communicator are custom derived data-types and the Cartesian topology in the `initialise` subroutine call. While the exact structure of the topology is left to MPI to compute, the program will exit with an error should the number of dimensions selected in any direction not be a factor of the size of the image in that direction: this is discussed further in Limitations.

The derived data-types underpin the successful execution of the software. Four such derived types are defined: two sub-array types, created using the `MPI_CREATE_SUBARRAY` and `MPI_CREATE_RESIZED` subroutines, and two vectors, using the `MPI_TYPE_VECTOR` call. The sub-arrays enable the use of `MPI_SCATTERV` and `MPI_GATHERV` routines for the distribution and collection of the original image to all working processors, as in Figure 1. The first sub-array type, `master_type`, allows for the initial distribution of the image data array onto the working processors. The second type, `block_type`, defines an $(M_p + 2)$-by-$(N_p + 2)$ array of the main M-by-N image for each working processor, and is used to gather the individual sub-arrays for each processor for image re-assembly.

Two vectors are also created. The first is a horizontal vector, which allows for rapid halo swapping between processors aligned above and below each other in the topology, and the second a vertical vector, which allows for rapid halo swapping between processors aligned to the left and right of each other in the topology.

Once the initialisation routine has completed, the processor on rank zero in the new, Cartesian communicator reads the input file, which is determined from the appropriate `fname`, `M` and `N` variables in the `problem_constants` module file. A call to `MPI_SCATTERV` is then performed to send a `master_type` sub-array to each of the P processors.

The Jacobi iteration then occurs. Each processor has a dependency on data held by other processes; as a result, every iteration of the Jacobi algorithm must be preceded with a series of non-blocking sends and receives to exchange the required data for every processor. Should blocking sends and receives be used, it is possible for the program to hang with nodes unable to progress while they wait for their counterpart – also waiting to receive – to send.

The difference between two most recent iterations is computed every `check_int` iterations, adjustable in the `problem_constants` module. When this value converges to less than the tolerance specified in the same module file, iteration is completed.

After computing their local subset of the image, each processor must then transmit its portion to all other processors to re-assemble the re-composed image. By using the `MPI_GATHERV` subroutine with the `block_type` sub-array, the image can be gathered on all processors rapidly.

The zeroth-rank processor will then write the image to the specified output file.

Testing

Testing parallel applications is often difficult, and this is exacerbated with the time restrictions in this work. However, a small suite of tests was performed on the final algorithm.

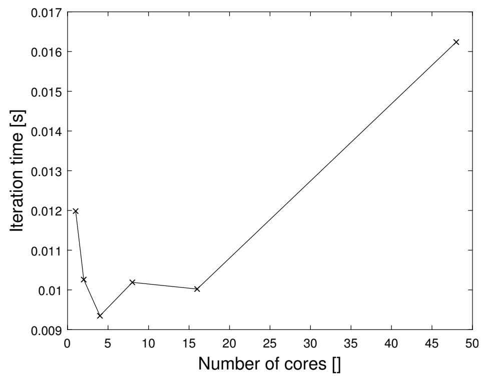
In particular, the results from every image included in the case study were cross-checked and verified against results produced using the sample solutions provided by the EPCC for the workshop, first by eye, and then also using a hybrid Bash/Python testing suite. The parallel applications were run for each possible case of image – automated through use of pre-prepared `problem_constants` module files – and the output read into a Python script running OpenCV. Through use of the `imread` function, the average value of the solution produced using both the sample solutions and the two-dimensional implementation were checked against each other. Owing for differences in the results produced by noise, and the additional halo swapping, average values within 5% of each other were taken to be coherent.

Further tests were performed to check that the program correctly exits in the case of incorrect input. In this case, the program should exit with an `ERROR STOP` should the incorrect number of processors be specified; the incorrect image dimensions specified; or if the dimensions of the topology are not factors of the dimension of the image.

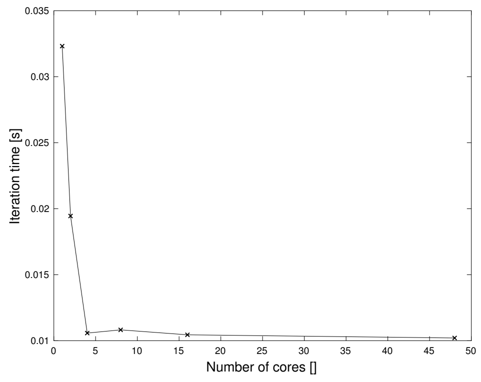
Data Gathering

The MPI standard provides the `MPI_WTIME` subroutine, which can be used in conjunction with the `MPI_BARRIER` construct to calculate wall-times for parallel programs. In this work, all programs were run ten times and an average time was taken.

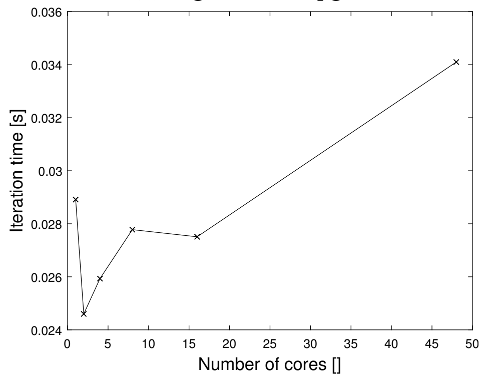
Five performance metrics were calculated to measure the performance of the algorithm across different core counts: the time to scatter the image data to all processors; the time to perform the main Jacobi iterations for each thread; the average time to perform the halo swapping routines; and



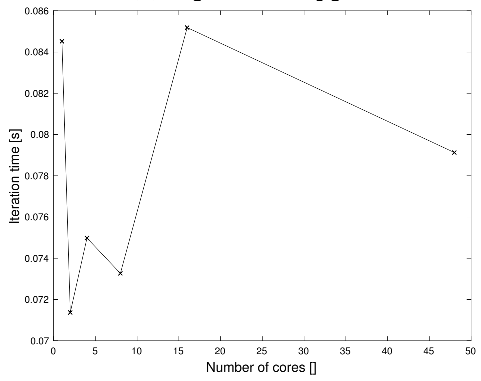
(a) `edge198x162.pgm`



(b) `edge256x192.pgm`



(c) `edge512x384.pgm`



(d) `edge768x768.pgm`

Figure 2: Iteration time as a function of the core count for various image sizes. Note that we expect the iteration time to decrease, since the size of the local sub-array for each worker is decreased.

the time required to gather the sub-arrays from the worker processes. From these timings, the time required per iteration can be computed, and the serial and parallel performance can be identified.

Results

The time required to complete the Jacobi iterations is presented in Figure 2; note that the heuristics of the time required per iteration is the same, since all images take the same number of iterations to compute (1300), so these have been omitted for the sake of brevity.

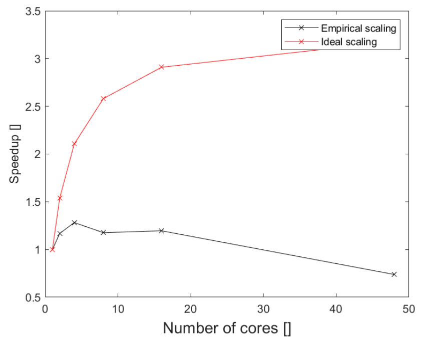
In addition, Figure 4 provides information as to the amount of time the software spends in each portion of its source code.

Figure 3 shows the strong scaling of the problem; while we appear to gain better than ideal scaling for many of the images, this is as a result of less than ideal performance for a processor count of 1.

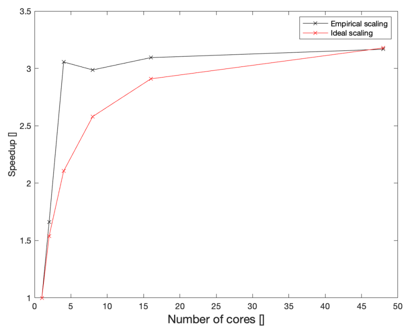
Discussion

Much of the results are as expected, although discrepancies have emerged.

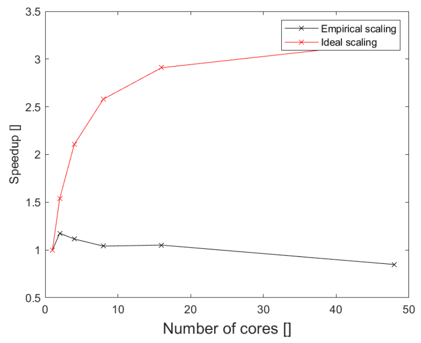
When considering Figure 2, we expect to see a quadratic decrease in iteration time, as the local sub-array size decreases in both M_p and N_p with increasing processor count P; it is evident that this doesn't occur.



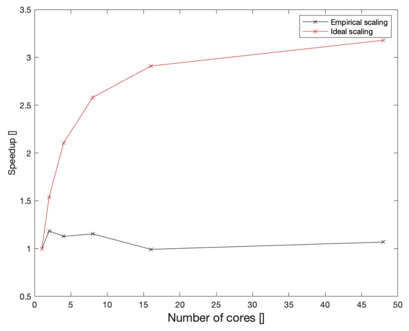
(a) `edge198x162.pgm`



(b) `edge256x192.pgm`



(c) `edge512x384.pgm`



(d) `edge768x768.pgm`

Figure 3: Ideal scaling against empirical scaling: note that we gain (seemingly) greater than ideal scaling as a result of poor performance for $P = 1$.

This effect is produced from the time required to send the data between processes, which is reported by the profiler to dominate iteration times for all images across all core counts, with data sending accounting for up to 40% of iteration timings. As is expected, the send time increases with core count for all images, with the difference in timings due to varying image sizes being largely insignificant (the bandwidth for ARCHER reported by a simple ping-pong code is higher than even the total volume of data being sent). It is therefore possible that the iteration performance is a trade-off between the time required to send the data between all processors, and the increase in performance made by having each worker compute a progressively smaller chunk of the overall image.

Figure 4 lends some weight to this theory: while the send and receive methods will not produce the same timings as the `SCATTERV` and `GATHERV` constructs, it is clear that the amount of time spent performing data transfers, as a proportion of program run-time, increases with core count. We could therefore expect, particularly for the smaller images where the relatively fixed sending overhead will dominate performance, that increasing core counts may be accompanied by a counter-intuitive increase in iteration time. This does indeed appear to be the case in (a) and (c) in Figure 2.

While this may begin to explain the trends in the data presented in Figure 2, it does not explain the large increases in execution time present for 4 and 16 cores in (d) in the Figures. Initial assessment would categorise these points as outliers and discard them accordingly, but multiple runs of the program, all on the same node, were performed to be able to reject 'outlier behaviour' in the data.

Further profiling would be appropriate to locate the source of the deviation from expected behaviour, but given that (d) provides the timings for the largest image, it would not be unsurprising for the algorithm's behaviour to be different: here, the iteration time is likely to be far higher compared to data sending (which will, in itself, be higher than normal owing to the greater number of data contained within the horizontal and vertical vectors used for the halo swapping.)

It is also not unreasonable for the performance to be characterised by the frequency of convergence checks; since this operation requires multiple reduction operations, the greater the number of convergence checks, the greater the iteration time. For all of the run-times presented here, the check frequency was kept constant. However, for larger core counts it is again likely that there will be a far greater influence from this step on the overall computation time.

The strong scaling presented in Figure 3 – that is, the change in performance for a fixed total problem size with increasing core counts – shows the scaling is generally very poor. The ideal scaling, computed using Amdahl's law with a semi-empirical value of $p = .7$ and s equal to the core count, is superimposed on each of the figures.

It is clear that the performance shown by the algorithm is sub-standard. It is important to note that performance appears to increase beyond ideal as a result of poor performance from the case where $P = 1$. There are a number of possible reasons why the scaling is not as expected: many of them have been discussed above. In general, while 70% of the algorithm is parallelisable, it is not possible for all of these portions to be parallelised, and the time taken to transfer halos between nodes, as well as the reductions for the convergence checking, will all prevent the realisation of ideal scaling. Data transfers also take finite time to compute, which is not assumed in Amdahl's law.

Interestingly, however, we do recover generally good scaling for the `256x192` image, despite the results of the other images. It could be that this image happens to strike the correct balance between the image processing and the MPI overheads, but the exact reason is unclear. After two cores, however, the speedup for this image is also limited. Again, proper profiling would allow the reasons behind this to potentially be uncovered.

Discussion: Send and Receive Methods

When considering the performance of the algorithm, it is also important to justify many of the algorithmic decisions made when designing the program.

In this case, it is important to discuss the send and receive methods implemented. The 'final' version of the program uses `MPI_Issend` and `MPI_IRecv` for non-blocking communication. However, one other sending option exists: `MPI_Isend`. Both are non-blocking – which is appropriate for their use here, as stated earlier – but the difference between them lies in when they signal to MPI that the send buffer can be used again:

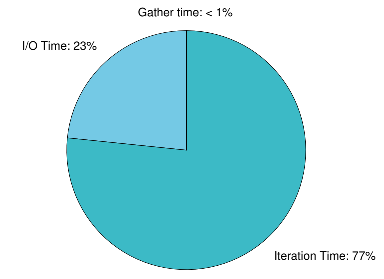
- `MPI_Isend` returns control either when the data has been copied locally in a buffer owned by the MPI library, or when the data has actually been sent;
- `MPI_ISsend` doesn't buffer data locally and control is returned only after the data has been transferred.

In any case, both methods were implemented and their performance measured. `MPI_ISsend` had lower execution times for all images, across all metrics, than `MPI_Isend`.

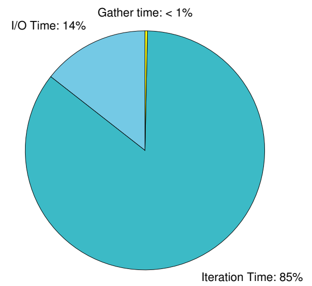
Potential Algorithm Improvements

The performance of the algorithm, while generally better with increasing core numbers, still leaves a lot to be desired. Changes to the algorithm, or the method used, could result in greater performance increases.

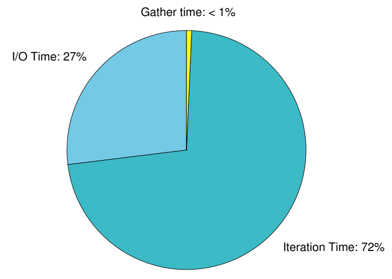
For example, modifying the problem so that there is no dependency on shared halo regions would remove much of the communication overheads present in the current program; this would most likely be performed by changing the underlying method used in the re-construction of the target image, such as by using a discrete Poisson solver (Pérez et al., 2016).



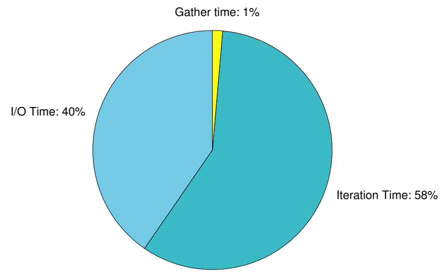
(a) `edge198x162.pgm`



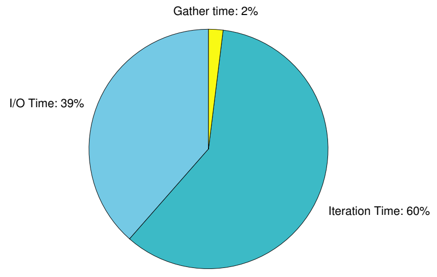
(b) `edge256x192.pgm`



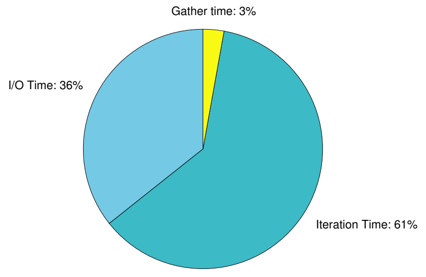
(c) `edge512x384.pgm`



(d) `edge768x768.pgm`



(e) `edge512x384.pgm`



(f) `edge768x768.pgm`

Figure 4: Time spent by each program in various portions of the code. Interestingly, while the iteration time will decrease for increasing core counts, the increased time to perform the initial scattering and final gathering, will hamper the overall performance of the algorithm.

Furthermore, it would be worthwhile to experiment with the use of different stopping criteria, in order to perform the least work possible: while the threshold for the difference between two iterations is adjustable in the `problem_constants` file, other convergence metrics, and ones which do not require the use of a global reduction, could be investigated. The current methodology, if applied carefully, may also still function when applied to a single chunk rather than by performing the reduction.

The use of MPI-I/O on a parallel file system (such as the one present on ARCHER) may also help to combat the ~30% of the program run-time spent in distributing and receiving the image data file. However, the use, and implementation of, MPI-I/O is outside the scope of this work.

The use of `MPI_ALLREDUCE` when computing the convergence checks also has a major impact on the performance of the algorithm; use of another, non-blocking method, may help to improve performance.

Limitations

The implementation discussed above also has multiple limitations: in particular, the program will not function properly if the number of nodes assigned in any Cartesian dimension is not a factor of the image size in that dimension.

It is also not possible to directly analyse the amount of time spent sending data and the iteration time together, since timing the data sends requires the use of a `MPI_BARRIER` construct which will arbitrarily slow the iteration time. Creating two different versions of the program to time these metrics separately does not guarantee reproducibility, with both the compiler potentially performing different optimisations, and the job management system assigning different nodes, with different bandwidths between them.

Conclusion

This report has investigated the performance of a two-dimensional Cartesian topology when applied to a set of sample images on the ARCHER supercomputing cluster. As can be seen with the results presented herein, there are many factors to consider when attempting to increase the performance of a parallel algorithm. In particular, performance is often a balance between gains due to parallelism, and the losses from their related overheads. The results here have shown poor scaling for sample images, although it is important to note that the results may be inappropriate when being considered at such short run-times. The processing of far larger images, which will naturally increase the processing times, may produce far more reliable and accurate timings than those presented here.

References

J. Canny. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(6):679–698, Nov 1986. ISSN 0162-8828. doi: 10.1109/TPAMI.1986.4767851.

Grant McFarland and Michael Flynn. Limits of Scaling MOSFETs. (January), 1995.

Patrick Pérez, Michael Gangnet, and Andrew Blake. Poisson Image Editing. Image Processing On Line, 5:300–325, 2016. ISSN 2105-1232. doi: 10.5201/ipol.2016.163. URL `http://www.ipol.im/pub/art/2016/163/?utm{_}source=doi`.

Irwin Sobel. An isotropic 3x3 image gradient operator. Presentation at Stanford A.I. Project 1968, 02 2014.