

Distributed Levenberg Marquardt (LM) Algorithm

1 – Introduction

Parameter estimation for function optimization is a well established problem in computing, as there are countless applications in practice. For this work, we will focus specifically in implementing a distributed and parallel implementation of the Levenberg Marquardt algorithm, which is a well established numerical solver for function approximation given a limited data set.

Parameter estimation is required for robotic applications, such as estimating relationships between multiple viewpoints in computer vision, also, orbit determination in aerospace applications requires function optimization using numerical methods based on limited observations. However, there are multiple other applications such as signal processing, control theory, and telecommunications that require such solutions. The Levenberg Marquardt algorithm is an iterative solver that performs calculations to improve the estimate at each step, the iterations continue until an adequate bound is reached. This iterative nature will allow us to implement a distributed version for parallel systems through load balancing at each step. Although this method has demonstrated to be robust and efficient, when solving high-order non-linear systems, the process might diverge, since convergence is not guaranteed, and the initial parameter estimate becomes critical for convergence. These technical details about the Levenberg Marquardt algorithm are essential when working with this technique for either the basic traditional single processor algorithm, or when working with a large dataset that requires a large CPU cluster with a distributed implementation. Initially, our most basic requirement is to implement a distributed algorithm that is consistent with the single CPU approach.

2 – Related work

Coleman and Plassmann [2] have previously also implemented a distributed version of the Levenberg Marquardt algorithm, where they also utilize the same framework, and also utilize the same functions to implement the LM algorithm such as QR factorization to solve the system of the Jacobian matrix. However, they attempted larger problem sizes that include up to $n=400$ parameters and $m=1600$ observation data points. However, they were able to successfully achieve speedup using an optimized QR Solver.

Furthermore, Suri et. al [3] also analyze the potential in implementing a distributed version of the Levenberg Marquardt algorithm for image processing applications, where they focus on the application using the MPI implementation of the LM algorithm.

3 – Levenberg Marquardt (LM) Algorithm

Although the LM algorithm is an iterative solver for parameter estimation for function optimization, LM still operates in a deterministic fashion, where identical results are produced if the data and input parameters are not changed. This deterministic property will allow us to test and verify the distributed implementation since we expect the same results for the same configuration.

The LM algorithm is based on the Gauss-Newton Method, where there are similar concepts in the frameworks, except the LM approach incorporates a dampening parameter that increases stability and therefore helps with convergence. Furthermore, the LM Algorithm is also similar to other numerical solvers such as gradient descent or the conjugate gradient method, which are also iterative solvers that improve the estimate at each step by selecting the parameters that are in the direction of the basis opposite to the gradient, which is in the direction that that will reach the optimum point. However, these methods are data dependent, and require symmetric positive definite systems for convergence.

The Levenberg Marquardt algorithm performs a curve fitting on a given dataset, by finding the optimum function parameters β based on a user specified model f , such that the final parameters can characterize the target function by minimizing the residual, which is the same as solving the least squares problem, where we want to minimize the sum of squares between the target values y_i and the output of the user model f with the input values x_i and the estimated parameters β with m number of input data points.

$$\arg \min_{\beta} \sum_{i=1}^m (y_i - f(x_i, \beta))^2$$

The algorithm requires an initial parameter estimate β_0 , which may be critical in convergence if the user model is high dimensional or non-linear with a large number of parameters, n . At each step, we update the estimated parameters β by a small amount in the optimum direction by adding this update delta values, δ , such that:

$$\beta_{t+1} = \beta_t + \delta$$

To find the update delta value, δ , we need to solve for the approximation of the sum of squares function by setting the gradient equal to 0

$$\mathbf{J}^T [y - (f(x_i, \beta) + \mathbf{J}_i \delta)] = 0$$

$$\mathbf{J}^T [y - f(x_i, \beta)] - \mathbf{J}^T \mathbf{J}_i \delta = 0$$

$$\mathbf{J}^T \mathbf{J}_i \delta = \mathbf{J}^T [y - f(x_i, \beta)]$$

Where \mathbf{J} is the Jacobian Matrix, or gradient, of the function that describes the user's model. And from this equation, we can directly calculate the delta values at each step by solving this set of linear equations in the form of $\mathbf{Ax}=\mathbf{b}$, where \mathbf{A} is a square matrix, \mathbf{x} , is our delta values vector, and \mathbf{b} is also a known vector of the same length as \mathbf{x} .

However, the Levenberg Marquardt algorithm also adds a regularization parameter, λ , that helps as the dampening factor, to produce the final LM equation that finds the delta values at each step:

$$(\mathbf{J}^T \mathbf{J}_i + \lambda \text{diag}(\mathbf{J}^T \mathbf{J}_i)) \delta = \mathbf{J}^T [y - f(x_i, \beta)]$$

This LM equation is still a system of linear equations of the form $\mathbf{Ax}=\mathbf{b}$, that involves additional computations to preprocess all the data to arrive at the compact form $\mathbf{Ax}=\mathbf{b}$. However, once at that form, we can use direct solvers to solve for our delta values at each step.

4 – Parallel implementation

The parallel implementation of this Levenberg Marquardt algorithm involves distributing the computations among a multi-processor cluster to theoretically reduce the execution time. Such performance is desired with large datasets, particularly when the function optimization involves a user specified function model with a large number of parameters, and when the input dataset is sufficiently large.

The approach for implementing a distributed LM algorithm was to modify the each of the multiple functions in the single CPU version. These functions to be distributed include matrix and vector operations such as QR factorization, and Matrix-Vector Multiplications.

The original single CPU version was obtained from Joachim Wuttke, as an open-source lmfit package.

Subsequently, each of the functions, including QR factorization were distributed on an IBM multi-core cluster utilizing the OpenMP framework as a shared memory distributed environment.

However, since the LM algorithm is an iterative method, and the solution is updated at each step based on the previous step, there is no way to distribute the iterations, we can only optimize and distribute the work computed at each step.

5 – Results

Wuttke's lmfit implementation of the LM algorithm in C [1] was modified to include the multi-core support for a distributed system using OpenMP. The algorithm was tested on Texas A&M's supercomputing cluster on the Hydra system, which is an IBM p5-575 cluster running AIX 5.2 operating system. Since OpenMP requires shared memory, it can only utilize one shared memory node at a time, and each node consists of 16 Power5+ processors running at 1.9GHz.

The Distributed LM algorithm was tested on a dataset that consisted of 500 datapoints, while attempting to fit a 50-degree order function which is a user model that involves the estimation of 50 parameters.

The test data was generated from the function that described as:

$$f(x, \beta) = \sum_{i=1}^{50} (-1)^i \cdot \beta_i \cdot x^i$$

Where the true parameters used to generate the test case were randomly chose as:

$\beta = [0.44082 \ 0.6298 \ 0.72711 \ 0.8682 \ 0.63953 \ 0.24975 \ 0.68518 \ 0.095737 \ 0.014262 \ 0.69496 \ 0.47007 \ 0.49305 \ 0.45147 \ 0.60947 \ 0.030277$
 $0.97042 \ 0.94414 \ 0.12785 \ 0.55289 \ 0.29387 \ 0.024763 \ 0.87836 \ 0.35353 \ 0.68657 \ 0.2226 \ 0.78852 \ 0.91811 \ 0.43532 \ 0.87149 \ 0.43578]$

Which produces the following function on the range between 0 and 1:

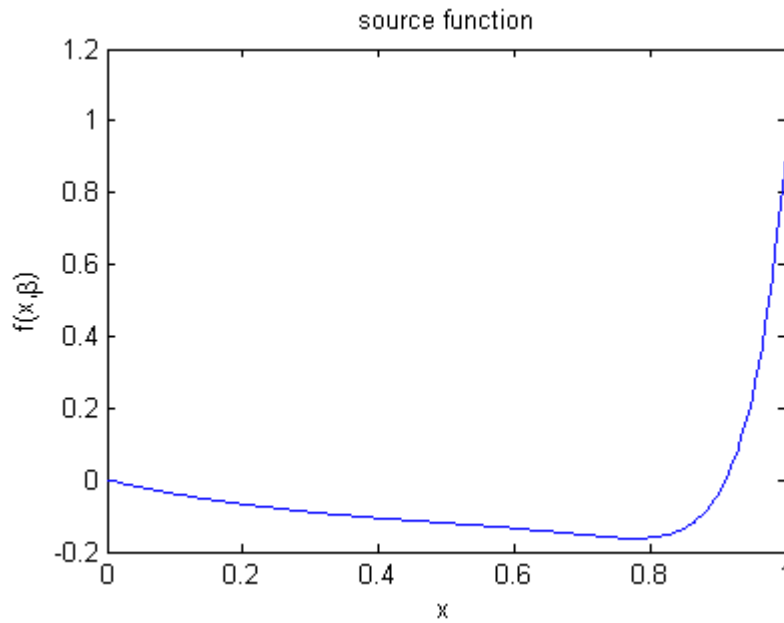


Figure 1. source function to generate the test data.

Subsequently we generated 500 random points between the range of 0 and 1 and evaluated our function $f(x,B)$ to generate our (x,y) input pairs, the following figure 2 illustrates the input test data for the LM algorithm, along with the source function:

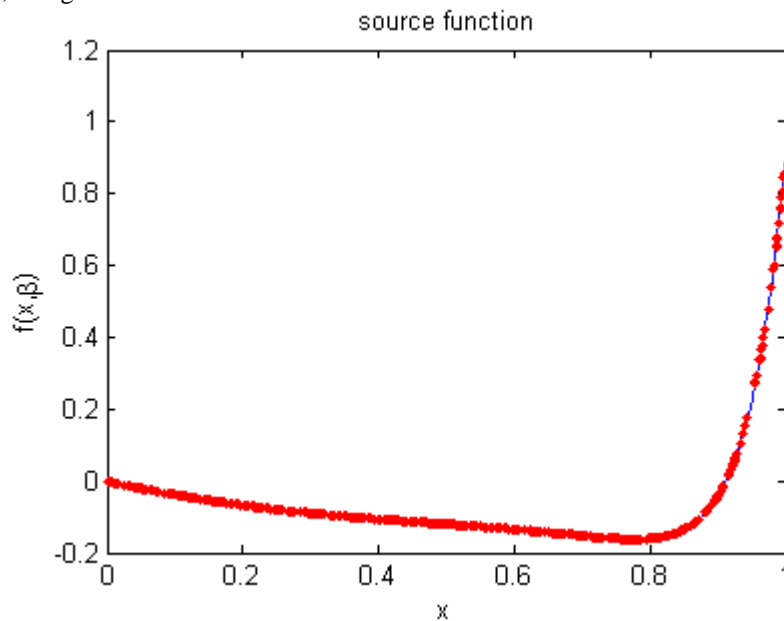


Figure 2. source function (blue trend line) along with randomly chosen input datapoints (red dots)

Next, we illustrate what our function returns when evaluated at the initial step of the LM algorithm with all function parameters β set to -1, and also the final output result of the function evaluated with the estimated parameters from Levenberg Marquardt.

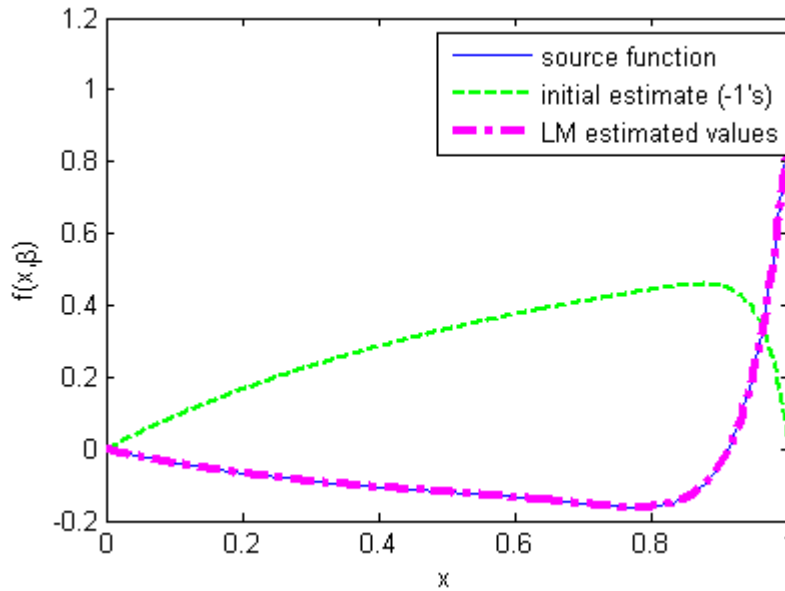


Figure 3. Source function (blue line) and initial estimate of model with parameters set to -1 (green dashes), and the output of the LM algorithm of the estimated function shown (pink dashes), which match the source.

These results as illustrated above, demonstrate the correctness of the LM algorithm, and also demonstrate the correctness of the distributed implementation, as all instances of the program return the same result and output regardless of the number of CPU's, as expected due to the deterministic nature of the LM algorithm.

Next, we demonstrate the performance and the speedup as we increase the number of CPU's, however, initially, we do not find any speedup as we increase the number of CPU's in the overall performance of the distributed LM algorithm.

Performance - Execution Time

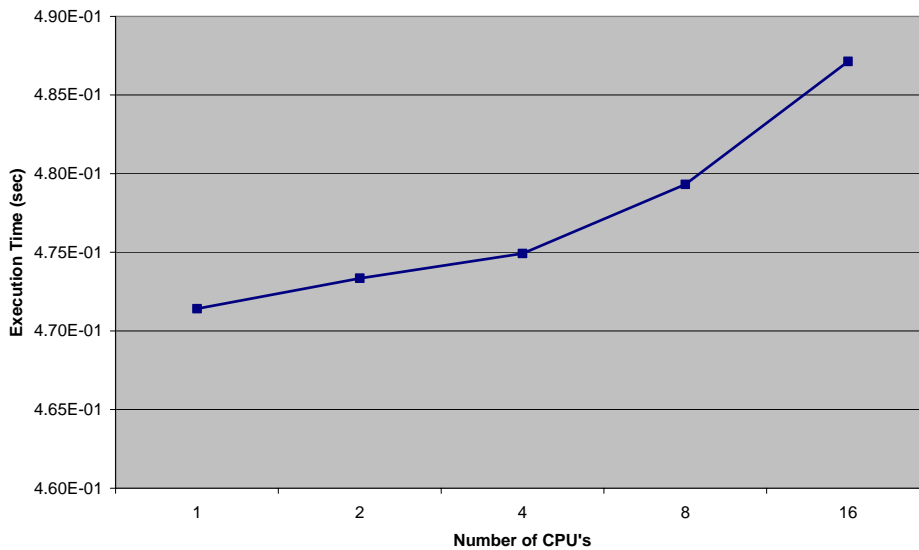


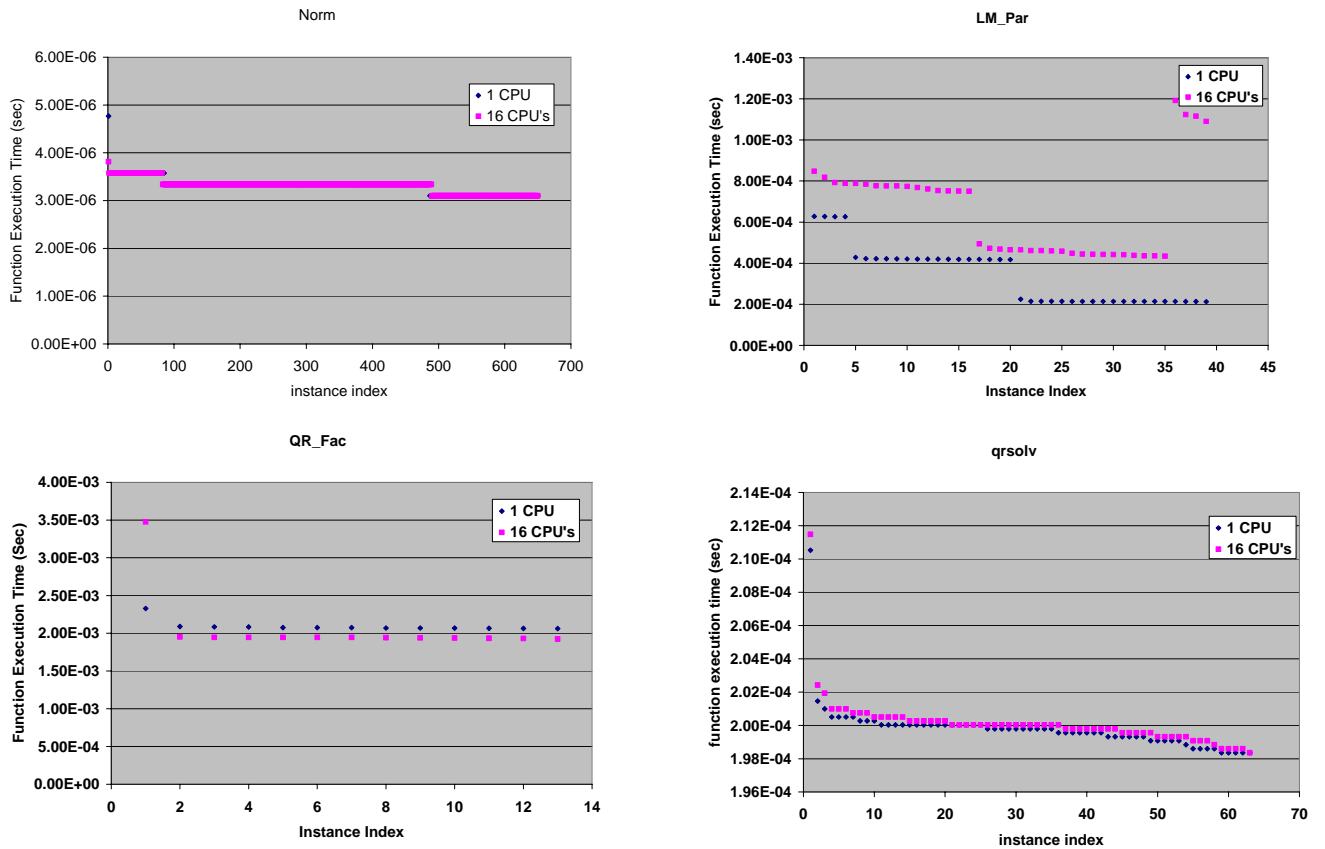
Figure 4. Performance of the Distributed LM implementation. Execution time in Seconds

However, as we can observe from the figure 4, the performance is actually degrading, as execution time increases as we increase the Number of CPU's.

Number of Processors	Execution Time
1	4.71E-01
2	4.73E-01
4	4.75E-01
8	4.79E-01
16	4.87E-01

Table 1. Performance. Execution time as we increase the number of CPU's

In order to further investigate the source of the slowdown, we evaluated the performance of each of the functions inside the LM implementation.



Where as we can see from the above figures, the first graph illustrating the performance of the NORM function is not affected by the number of CPU's, as we did not optimize or distribute the norm function. Next, the LM_PAR function shows a significant decrease in performance as we increase the number of CPU's. The only function where we were successfully able to increase performance was the QR_FAC function, but since the LM_PAR calls multiple instances of QR_SOLV, where the performance slightly decreased, this caused LM_PAR to degrade as the CPU's increased.

6 – Conclusions

As we have observed, we have demonstrated a successful distributed implementation of the Levenberg Marquardt algorithm. We have proven correctness on our parallel implementation, and we have demonstrated a theoretical enhancement to the single CPU implementation.

However, the decrease in performance as we increase the number of CPU's could be explained by the lack of sufficiently large data. Since we only tested on one case of data, with 500 data points and 50 parameters, the overhead of the OpenMP might have caused the slowdown during execution. However, increasing the number of parameters, or data points compromised the stability of the convergence of the LM algorithm. Consequently, our implementation demonstrated a parallel implementation with the potential to enhance performance as the problem size increased.

7 – Future Work

Our results for this work have shown that additional efforts are needed to further investigate the increase in execution time of the distributed implementation of the Levenberg Marquardt algorithm. For example, additional test cases could be reviewed to vary the problem size with larger number of parameters.

Additional enhancements could be made to the solver of linear equations, where other solvers could be considered instead of the QR Factorization approach that would allow better performance for a distributed implementation.

8 - References

- [1] J Wuttke, LMFIT C Implementation, <http://messen-und-deuten.de>
- [2] Coleman, T. F. and Plassmann, P. E. 1992. A parallel nonlinear least-squares solver: theoretical analysis and numerical results. *SIAM J. Sci. Stat. Comput.* 13, 3 (May. 1992)
- [3] N. N. R. Ranga Suri, Dipti Deodhare, P. Nagabhusan, Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters - A Case Study , *ICVGIP 2002, 3rd Indian Conference on Computer Vision, Graphics and Image Processing*, Ahmadabad

Appendix – A: Compile and Run Commands

Compile Command:

```
xlC -q64 -qsmp=omp -O3 -o lmtest.exe lmin.c lm_eval.c lm_test_f.c -lm
```

Run Command – Sample Batch Job File Script:\

```
#@ shell = /bin/ksh
#@ comment = 8-proc OpenMP job
#@ job_name = finalomp_8.job
#@ error = $(job_name).o$(schedd_host).$(jobid).$(stepid)
#@ output = $(job_name).o$(schedd_host).$(jobid).$(stepid)
#@ job_type = parallel
#@ resources = ConsumableCpus(8) ConsumableMemory(5mb)
#@ wall_clock_limit = 00:00:10
#@ node = 1
#@ total_tasks = 1
#@ notification = always
#@ class = cs_group
#@ queue
export OMP_NUM_THREADS=8
./lmtest.exe
/usr/local/bin/jobinfo
```