An Overview of Parallel Methods for Unconstrained Optimisation

Tim Oliver
Centre for Mathematical Software Research
University of Liverpool

January 30, 1990

Abstract

This paper presents an overview of parallel algorithms for unconstrained optimisation. The suitability of algorithms to the PUMA architecture are discussed and the difficulties involved in combining within an algorithmic cost model the high level function evaluation cost with low level communications cost. Some sections of the linear algebra are highlighted as requiring further study to develop more efficient parallel optimisation algorithms.

1 Introduction

The broad field of numerical optimisation is usually split into sub-categories according to certain properties of the problem to be solved. Such a classification is useful because it is usually impractical, and certainly inefficient, to solve a specific problem using a general optimisation routine. By identifying categories of optimisation problem, account can be taken of special features of the problem leading to a more efficient optimisation routine. A convenient classification is based on the properties of the objective function, $f(x)$, and the constraint functions. This leads to a large but manageable number of categories including linear functions, smooth nonlinear functions and non-smooth nonlinear functions with or without similar categories of constraint functions. See [7] for an excellent summary of numerical optimisation. Most of the recent research into parallel algorithms has concentrated on the problem of unconstrained optimisation of smooth nonlinear functions. This category of problems is fundamental to numerical optimisation, and its relative simplicity permits easier investigation of the principles of parallel optimisation algorithms. For these reasons we restrict ourselves in this paper to considering this class of functions.

The unconstrained optimisation problem is of the form:

$$\lim_{x \in \Re^n} f : \Re^n \rightarrow \Re$$

where $f(x)$ is at least twice continuously differentiable. Sequential unconstrained optimisation algorithms are usually based on the following algorithm:

Model Algorithm Let x_k be the current estimate of x^* the minimum of the function $f(x)$.

Step 1 *Test for convergence.*
Terminate the algorithm if the convergence conditions are satisfied returning x_k as the solution.

Step 2 *Compute a search direction.*
Compute a vector p_k to use as a search direction.

Step 3 *Compute a step length.*
Compute a scalar α_k such that $f(x_k + \alpha_k p_k)$ satisfies the condition
$f(x_k + \alpha_k p_k) < f(x_k)$.

Step 4 *Update estimate for the minimum*.
Set $x_{k+1} = x_k + \alpha_k p_k$, $k = k + 1$ and go to Step 1.

Computing the solution to an optimisation problem is often extremely costly. This is partly due to high costs in evaluating the objective function, $f(x)$, which may, for example, be a complex simulation or a solution of a system of equations. Most algorithms also use higher derivative information, such as the gradient vector, $g(x)$, and perhaps the Hessian matrix, $G(x)$, to improve convergence. The most successful and efficient sequential algorithms use a *quadratic model* of $f(x)$:

$$f(x_k + p_k) \approx f(x_k) + g_k^T p_k + \frac{1}{2} p_k^T G_k p_k$$

where g_k is the *gradient vector* and G_k the *Hessian matrix* of $f(x)$ at the current point x_k. When $f(x)$ is expensive, the gradient vector and Hessian matrix will also be expensive and may not be available analytically. In this situation finite-difference approximations to higher derivatives may be used. For example, the gradient vector at the current point, x_k, may be approximated using forward differences by:

$$g(x_k)_i = \frac{f(x_k + h_i e_i) - f(x_k)}{h_i}$$

where h_i is the stepsize and e_i is the ith unit vector. When the problem size is not too large ($n \leq 100$, say) function evaluation dominates the overall cost. For much larger problems the linear algebra costs incurred in Step 2 becomes significant and this must be taken into account when developing parallel algorithms. We will consider mainly the case where $f(x)$ is expensive, although the ideas presented are still useful when the linear algebra costs are significant.

Two possible approaches for developing parallel optimisation algorithms are suggested by Byrd *et al.* [2]. The first is to utilise a standard sequential optimisation algorithm but evaluate $f(x)$ (and perhaps derivatives) with a parallel algorithm. This relies on the user providing parallel function evaluation routines which may not be possible. The second approach assumes the user provides sequential function evaluation routines and a parallel optimisation algorithm which calculates multiple function (and derivative) values in parallel is used. These two approaches are compatible and could be combined. This would be particularly profitable in a full `occam` environment where the user function processes could be instantiated dynamically on idle remote processors. As discussed in [2, p.169] this type of concurrency is well suited to MIMD architectures, both with local memory and shared memory.

To be able to assess the performance of parallel optimisation algorithms we need to select quantitative features of the algorithm which accurately reflect its cost. Usually, in sequential optimisation, two factors are specified:

1 the total number of *equivalent function evaluations* (one gradient evaluation is considered to be equivalent to n function evaluations), and

2 the total number of iterations.

For parallel optimisation algorithms we adopt the criteria used by van Laarhoven [20]:

1 the total number of *parallel equivalent function evaluations* (the total number of parallel steps, one parallel step being defined as up to p simultaneous function evaluations, where p is the number of available processors), and

2 the total number of iterations.

This parallel equivalent function evaluation is comparable with the *concurrent function evaluation step* defined by Schnabel [16] which assumes gradients are approximated by finite differences. Assuming that parallel equivalent function evaluations dominate the cost of the algorithm, Schnabel proposes a definition of speedup as:

$$\text{speedup} = \frac{\text{total number of equivalent function evaluations}}{\text{total number of parallel equivalent function evaluations}}$$

Efficiency is then defined in the usual manner:

$$\text{efficiency} = \frac{\text{speedup}}{p}$$

Given the assumption that the cost of evaluating $f(x)$ is large, these expressions provide a good indication of the expected performance of the parallel algorithms on real parallel machines. If, however, $f(x)$ is cheap and n is large then the linear algebra in Step 2 becomes significant and these expressions will no longer be valid.

Some of the earliest work on parallel optimisation was done by Chazan and Miranker [3]. They investigated the use of parallel searches in conjugate directions and did not need any gradient information. This work was extended by Sutti [18, 19] who considered both synchronous and asynchronous methods. He showed that the asynchronous counterparts to his synchronous algorithms converged to a unique fixed point, but this was not necessarily the minimum point. More recent work has looked at parallelising sequential algorithms which make use of derivative information. We examine the *Newton* and *quasi-Newton* family of methods which store either an explicit representation or an approximation of $G(x)$ in Section 2. The performance of proposed parallel algorithms is compared with their sequential counterparts and with one another. Section 3 discusses the *conjugate gradient* method which does not store $G(x)$ explicitly, and parallel algorithms which utilise conjugate search directions. Finally, we suggest areas for further research in Section 4.

2 Newton and quasi-Newton methods

This family of methods computes the search direction, p_k, in Step 2 of the model algorithm directly as the solution to:

$$G_k p_k = -g_k$$

In Newton's method G_k is computed explicitly either analytically or using finite differences. Quasi-Newton methods do not compute G_k directly but instead an approximation, B_k, to the Hessian is maintained and updated in Step 4 of the model algorithm. The most widely used update is the rank 2 *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) update:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k}$$

where $s_k = \alpha p_k$ and $y_k = g_{k+1} - g_k$. If an analytic Hessian is available, the Newton method is generally used for its superior convergence properties. When the analytic Hessian is not available the quasi-Newton method is used if function evaluation is expensive and the problem size n is not too large, otherwise if the function evaluation is cheap a finite difference Newton method is used.

Several papers have looked at methods for utilising parallel function and gradient evaluation. van Laarhoven [20] has examined Straeter's parallel variable metric algorithm [17]. During Step 4 this method computes in parallel the gradient at n points displaced by small steps from x_k along n linearly independent directions. These gradient values are then used to sequentially update the approximate inverse Hessian H_k by the symmetric rank 1 update. The resulting algorithm exhibits quadratic termination. van Laarhoven also develops a parallel generalisation of Broyden's rank 1 formula. In addition, he uses parallel function and gradient evaluation during the line search. His algorithm computes $f(x)$ at multiple points along the search direction and chooses a trio of points which bracket the minimum. The line search also incorporates a parallel interpolation procedure.

Freeman [6] points out that these algorithms are not guaranteed to produce positive definite matrices H_k. He then proceeds to describe a similar algorithm which utilises a parallel symmetric rank 2 updating formula based on Davidon's updating formula [5]. This algorithm has quadratic termination and guarantees that the approximate matrices H_k exist and are positive definite.

Although the preceding algorithms utilise parallel computation for gradient evaluation they do not allow the n matrix updates to be performed in parallel. This operation could then become the major cost within each iteration. These algorithms find the minimum of a quadratic function in one iteration, compared with

n iterations for a sequential quasi-Newton update algorithm. This is because each iteration of the parallel algorithms performs n quasi-Newton updates. Hence these algorithms show good performance through a much reduced number of iterations. However, each individual iteration has a greatly increased equivalent function evaluation cost, but this is mainly hidden by the use of parallel gradient evaluations. Nevertheless there is still scope for improved performance. Only a low level of parallelism is achieved in the line search step—most processors remaining idle, and as already mentioned the updates must be performed sequentially—all extra processors being idle.

The normal description and cost analysis of these methods assumes that at least $n + 1$ processors are available to compute analytic gradients or $(n + 1)^2$ processors if gradients are approximated by finite differences. If only $n/2$, say, processors are available then the algorithms in their proposed forms cannot be executed, also any processors available in addition to those required cannot be utilised. This highlights a general problem for any parallel optimisation routine, namely that the grain size of the parallel algorithm will be determined directly by n.

Many of the optimisation test functions provide good examples of this difficulty. Rosenbrock's [12] function in 2 variables ($n = 2$) with analytic first derivatives can use at most 3 processors to calculate the gradient values used to update the inverse Hessian approximation. If gradients are approximated by finite differences then 9 processors could be utilised. Hence, even if the parallel array contained many more processors the maximum speedup possible would be less then 9 since only that number of processors could be used by the algorithm. A further point follows from this. The expense of function evaluation, although dependent on n, can be considerable even for small n since the function itself may be, for example, a sub-optimisation problem or a simulation. Thus a problem function with large cost but only small dimension n which requires an impracticably large run time to solve on conventional computer architectures can only have the solution time reduced by a small factor of order n or perhaps n^2 but not p. Of course, many small dimension problem functions, including the Rosenbrock function, are very cheap to solve and in these cases the parallel methods would probably have poorer performance than their sequential counterparts due to the high communication to computation ratio.

A well known extension to Rosenbrock's function gives a general function of dimension n (n even). For a larger dimension problem of, say, $n = 20$ either 21 or 441 processors could be utilised. However, it is likely that processor arrays would have some intermediate number of processors. If an analytic gradient algorithm is executed only 21 processors are used and the maximum speedup attainable is only 21. The remaining processors not used by the algorithm could be made available to other tasks running on the array, requiring multi-user operating system support on the machine. To make use of all the available processors on the array executing a finite difference gradient algorithm could be achieved in two ways. Firstly, the 441 processes required by the algorithm could be mapped onto the smaller number of available processors. This use of *excess parallelism* would allow the maximum possible speedup given the limited resources. A second approach would be to develop algorithms which compute only as many of the gradient vectors as the number of available processors allows.

We now describe a family of parallel methods due to Byrd, Schnabel and Shultz [2] which follow this approach and may use a variable number of processors. These algorithms are interpolations between Newton's method and a quasi-Newton method. During the line search, Step 3 of the model algorithm, these methods utilise idle processors to calculate gradient information which is then used in Step 4 to update the Hessian approximation B_k. As in the previous algorithms, the use of gradients to improve the Hessian approximation is expected to decrease the number of iterations required. In their papers Byrd *et al.* consider the case where gradient vectors are computed by finite differences, but the same ideas may be applied when gradients are computed analytically with only the number of processors utilised changing.

The algorithms due to Straeter and Freeman compute the gradient information in Step 4 of the model algorithm adding an equivalent function evaluation to the cost of each iteration. In contrast, the algorithms due to Byrd, Schnabel and Shultz do not add any equivalent function evaluation cost to each iteration since the gradient information is computed in parallel with the line search of Step 3 and only as many function evaluations as the number of processors permits are computed. Hence full use is made of the available processors during Step 3 without increasing the run time. Schnabel calls this process *speculative gradient evaluation*.

There are alternative uses of the processors during the line search step. As already mentioned, van Laarhoven uses multiple processors to bracket the minimum and compute the interpolation. Lootsma [11]

and others have also suggested evaluating $f(x)$ at multiple points in the search direction and even in other directions in parallel with the computation of $f(x)$ at the trial point. However, it does not seem likely that points off the search direction will be better choices for the next iterate than those on that line. Also the large body of results from sequential quasi-Newton methods clearly show that on average only between 1 and 2 trial points are selected before an acceptable point is found and at each new accepted point a complete gradient evaluation is performed. This ratio of function evaluations to gradient evaluations suggests that processors would be much better utilised in computing a speculative gradient than in computing additional trial points when the current trial point is likely to be accepted anyway.

Schnabel [16] makes some suggestions about the use of the gradient information computed at a rejected trial point. One simple approach would be to use the search direction gradient to help compute the step length to the next trial point along the current search direction. Some current sequential line search algorithms use gradients at trial points but these do not improve the algorithm's performance significantly. Another suggestion is to use gradient information to solve a tensor model of the function rather than the standard quadratic model to find the next iterate [15]. Finally he proposes updating the Hessian G_k immediately using the gradient information at the rejected trial point and computing a new search direction.

If the number of processors available p is less than or equal to $n + 1$ then Byrd *et al.* recommend using the quasi-Newton algorithm with $p - 1$ elements of the gradient vector at the trial point speculatively computed in parallel with the trial point function evaluation. When a trial point is accepted the remaining elements of $g(x)$ are computed in parallel, otherwise a new trial point is chosen and the process repeated.

When $p \geq (n^2 + 3n + 2)/2$ then the entire Hessian matrix can be approximated by finite differences along with the trial point function and gradient in one concurrent function evaluation step. This method is a parallel discrete Newton method. More than $(n^2 + 3n + 2)/2$ processors cannot be utilised.

The most common situation will be $n+1 \leq p \leq (n^2+3n+2)/2$. In this case $m = \lfloor (p-(n+1))/(n+1) \rfloor$ gradient vectors from around the trial point are computed in Step 3 of the model algorithm. In [2] they describe and compare 11 such algorithms including the quasi-Newton and discrete Newton algorithms mentioned above. These algorithms all include at Step 4 a multiple update of the Hessian approximation G_k using the m gradient vectors along the finite difference directions. The methods fall into 3 categories based on the use made of the standard quasi-Newton update in the search direction:

1 Only the gradients along the finite difference directions are used to update G_k.

2 The search direction update of G_k is performed after the finite difference updates.

3 A temporary update of G_k in the search direction is made after the finite difference updates, and the resulting matrix is used to compute the search direction for the next iteration.

Two different methods are used to select the finite difference directions. One method simply cycles through the unit vectors and the other computes a set of m directions that are orthogonal to the previous $m - 1$ directions. Also, both the BFGS and PSB update formulae are used. The first and third categories of methods have m-step quadratic convergence and 1-step Q-superlinear convergence rates. Neither of these results has been shown for the second category in general.

Computational results are presented for the case $m = 1$. These show that the first category of methods all perform more poorly than the parallel quasi-Newton method using BFGS updates even though they use finite difference Hessian information. Temporary search direction update methods gave better performances but were still not better than the BFGS method. The third category of methods using both finite difference and search direction updates has the best performance being significantly better than the BFGS method. Of the three methods in this category the two which utilised the BFGS update were superior and Byrd *et al.* recommend a unit vector finite difference, BFGS method since it does not have the complication of calculating the conjugate directions which are required by the other BFGS method.

In [1] simulated results are presented for the BFGS unit vector finite difference algorithm and the parallel BFGS quasi-Newton and Newton methods when $n = 20$. Over the range of problems tested the average speedup over the sequential BFGS method of the parallel BFGS quasi-Newton method was 17.5, and for the parallel Newton method the average speedup was 82.3. With m ranging from 1 to n, the BFGS unit vector finite difference algorithm gave average speedups between 32.6 and 69.5. These simulated speedups assume that each algorithm had sufficient processors to compute all the speculative gradient

evaluations in a single concurrent evaluation step. That paper also shows that the BFGS unit vector finite differences algorithm has superlinear convergence.

If function evaluation is not expensive the linear algebra involved in Steps 2 and 4 becomes significant. It is thus essential to parallelise these steps if possible to maintain good performance. In [1] four different implementations of the linear algebra are considered. This shows that an efficient algorithm can be developed by storing and updating G_k^{-1}, the inverse Hessian approximation, using only matrix-vector and rank 1 updates which parallelise well. They demonstrate this by implementing a parallel algorithm based on an unfactored G_k^{-1}, distributed by rows.

3 Conjugate direction methods

Sequential conjugate gradient methods are used when the Hessian matrix is too large to be stored explicitly. With MIMD machines with large amounts of local memory per processor, larger problems which previously were solved by a conjugate gradient method on a conventional sequential computer may now be solved by quasi-Newton methods on a MIMD computer.

The conjugate gradient method computes a search direction in Step 2 of the model algorithm as follows:

$$p_k = -g_k + \beta_{k-1} p_{k-1}$$

where:

$$\beta_{k-1} = \frac{y_{k-1}^T g_k}{g_{k-1}^T g_{k-1}} \quad \text{or} \quad \beta_{k-1} = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$$

The method is generally only linearly convergent, but large improvements can be achieved by preconditioning the matrix to reduce the number of distinct eigenvalues.

For a parallel architecture the conjugate gradient method does not have as much scope for parallelisation as quasi-Newton methods. However, Han [9] has suggested using parallel minimisations in conjugate subspaces within a quasi-Newton method. In this algorithm the computation of the search direction, which in quasi-Newton methods is achieved by solving a system of linear equations, is split into 2 parts: first, m parallel minimisations are performed in m conjugate subspaces to give search directions $d_i, \quad i = 1, m$, then the summation vector of these directions is used as the search direction to find the next iterate x_{k+1}. Han uses the BFGS update for which he develops a method to update the conjugate subspaces so that they remain conjugate with respect to the BFGS updated Hessian approximation. In [14] Powell improves the conjugate subspace update allowing the Hessian to be stored as Cholesky factors. The algorithm aims to give better performance than the standard sequential quasi-Newton method by computing a more accurate search direction at each iteration. There are no computational results for this algorithm so its performance cannot be compared with the methods of the previous section. However, the method only uses the standard quasi-Newton update to the Hessian without using any more information that may be computed in the parallel step, and so its performance is probably poorer than the parallel BFGS finite difference quasi-Newton methods.

One area in which the conjugate subspace method has been applied is in partially separable optimisation problems [4, 10]. These problems have sparse Hessian matrices which makes it easier to compute and update the conjugate subspaces.

4 Conclusions

The research to date has emphasised parallelising existing sequential algorithms. This has been most successfully achieved in the work by Byrd *et al.* on quasi-Newton and discrete Newton methods. They show that parallel function evaluations, especially during the line search, allow for large reductions in execution time. However these algorithms are not as efficient as the sequential algorithms in terms of processor utilisation: if the parallel algorithms were sequentialised they would have a poorer performance than the existing sequential algorithms. This does not take into account the additional cost of communication on the PUMA local memory architecture. Fortunately the communications involved are only order n for each

function evaluation so this would not be too significant provided $f(x)$ was expensive. If $f(x)$ is cheap then communication becomes dominant and these algorithms would have very poor performance.

The performance of optimisation algorithms is measured in terms of function evaluation cost. As the real cost of the function in floating point operations will not be known there is no way of developing general algorithmic cost models in terms of low level floating point and communication primitives only. Hence the effect of the PUMA local memory architecture on parallel algorithms cannot easily be assessed quantitatively. However models in terms of both low level primitive operations for the linear algebra steps and high level function and derivative evaluations can be constructed. Specific operation counts (for function and derivative evaluation) of an example objective function, or 'average cost' objective function, can then be substituted into the general cost model to give an idea of the performance of the PUMA architecture. An example of such a model is presented in [13] which develops a parallel Newton method using analytic first and second derivatives.

Further work is needed to develop parallel methods for incorporating the additional derivative information computed in the BFGS finite difference quasi-Newton algorithm into the Hessian approximation and for computing the search direction. These linear algebra steps are particularly important when function evaluation is reasonably expensive but still does not completely dominate the algorithmic cost.

References

[1] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Parallel quasi-newton methods for unconstrained optimization. Technical report, Department of Computer Science, University of Colorado, April 1988.

[2] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Using parallel function evaluations to improve hessian approximation for unconstrained optimization. In Hammer et al. [8], pages 167–193.

[3] D. Chazan and W. L. Miranker. A nongradient and parallel algorithm for unconstrained minimization. *SIAM Journal on Control*, 8(2):207–217, May 1970.

[4] M. -Q. Chen and S. -P. Han. A parallel quasi-newton method for partially separable large scale minimization. In Hammer et al. [8], pages 195–211.

[5] W. C. Davidon. Optimally conditioned optimization algorithms without line searches. *Mathematical Programming*, 9:1–30, 1975.

[6] T. L. Freeman. Parallel projected variable metric algorithms for unconstrained optimisation. Technical report, Centre for Mathematical Software Research, University of Liverpool, September 25 1989.

[7] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[8] Peter L. Hammer, Robert R. Meyer, and Stavros A. Zenios, editors. *Parallel Optimization on Novel Computer Architectures*, volume 14 of *Annals of Operations Research*. J. C. Baltzer AG Scientific Publishing Company, 1988.

[9] S-P. Han. Optimization by updated conjugate subspaces. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis: Pitman Research Notes in Mathematics Series 140*, pages 82–97. Longman Scientific & Technical, Burnt Mill, England, 1986.

[10] M. Lescrenier. Partially separable optimization and parallel computing. In Hammer et al. [8], pages 213–224.

[11] F. A. Lootsma. Parallel non-linear optimization. Technical Report 89-45, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1989.

[12] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, March 1981.

[13] Tim Oliver. A parallel Newton method for unconstrained optimisation. PUMA working paper, University of Liverpool, 28 September 1989.

[14] M. J. D. Powell. Updating conjugate directions by the BFGS formula. *Mathematical Programming*, 38:29–46, 1987.

[15] R. B. Schnabel and P.Frank. Tensor methods for nonlinear equations. *SIAM Journal of Numerical Analysis*, 21:815–843, 1984.

[16] Robert B. Schnabel. Concurrent function evaluations in local and global optimization. *Computer Methods in Applied Mechanics and Engineering*, 64:537–552, 1987.

[17] T. A. Straeter. A parallel variable metric optimization algorithm. NASA Technical Note L-8986, Langley Research Center, 1973.

[18] C. Sutti. Nongradient minimization methods for parallel processing computers, part 1. *Journal of Optimization Theory and Applications*, 39(4):465–474, April 1983.

[19] C. Sutti. Nongradient minimization methods for parallel processing computers, part 2. *Journal of Optimization Theory and Applications*, 39(4):475–488, April 1983.

[20] P. J. M. van Laarhoven. Parallel variable metric algorithms for unconstrained optimization. *Mathematical Programming*, 33:68–81, 1985.