Int. J. Nonlinear Anal. Appl. 12 (2021) No. 1, 1123-1133
ISSN: 2008-6822 (electronic)
http://dx.doi.org/10.22075/ijnaa.2019.16979.1900

A family of parallel quasi-Newton algorithms for unconstrained minimization

M. S. Salim[a], A.I. Ahmed[a,*]

[a]Department of Mathematics, Faculty of Science, Al-Azhar University, Assiut, Egypt

(Communicated by Saman Babaie-Kafaki)

Abstract

This paper deals with the solution of unconstrained optimization problems on parallel computers using quasi-Newton methods. The algorithm is based on exploiting parallelism in function and derivative evaluation costs and linear algebra calculations in the standard sequential algorithm. A computational problem is reported for showing that the parallel algorithm is superior to the sequential one.

Keywords: parallel algorithm, unconstrained optimization, quasi-Newton
2010 MSC: Primary 65K05; Secondary 90C53.

1. Introduction

Optimization has many applications in all areas of computer science, mathematics, engineering, economics, medicine and statistics that can be modelled as optimization problem. Several studies have been reported by many researchers for decades for solving optimization problems [1, 7, 24, 26]. Quasi-Newton methods are a class of numerical methods for solving the unconstrained optimization problem

$$\min f(x), \quad x \in \mathbb{R}^n, \tag{1.1}$$

where $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is a nonlinear continuously differentiable real–valued function. The unconstrained optimization consists of determining the values that will minimize a function $f(x)$ of several variables. Since there are practical applications of the unconstrained optimization, a large amount of effort has been spent on the development of efficient serial algorithms for solving problem (1.1) [6, 13, 25, 37]. Parallel computing technology have made it possible to solve problem (1.1) with large number of variables and more rapidly and effectively.

*Corresponding author
Email addresses: m_s_salim@yahoo.com (M.S. Salim), aiahmed@azhar.edu.eg (A.I. Ahmed)

Received: January 2019 *Revised:* November 2019

In sequential optimization methods the expensive computational costs can be found in two main sources, the evaluation of $f(x)$ and its derivatives, and the linear algebra computations if the number of variables is large. For the cost from the evaluation of $f(x)$ and its derivatives, one can be parallelize each of these evaluations. The effectiveness of this technique depends on the availability of the parallel version of $f(x)$ and its derivatives, and how fully it parallelizes the evaluation. The linear algebra computations contain the calculation of the search direction, which include the update of the Hessian matrix or its inverse and finding the stepsize that may include one-dimensional optimization problem. Our interest here is the parallel extension to the quasi-Newton methods which are suitable for the solution of large-scale optimization problems. This approach will be appropriate whenever a good parallel implementation of $f(x)$ is available or not, and when the remaining costs of the optimization algorithm (linear algebra) are significant.

Since we are interested in performing different operations on different data in the same time, the proposed parallel algorithms need multiple instruction, multiple data (MIMD) computer. while we don't require a single instruction, multiple data (SIMD) computer that perform the same operation on different data simultaneously. MIMD computers can be classified according to their memory organization as shared memory where all processors share access to a global memory and distributed memory where each processor has its own local memory and there is no globally shared memory since they are connected across interconnection network . When a processor needs data from the local memory of the others to perform local computations, message passing has to be performed via the interconnection network. Both kinds are suitable for the algorithm described here, but it may be more appropriate to use shared memory MIMD computers when n is not large due to communication or synchronization costs.

There are many parallel unconstrained optimization algorithms such as non-derivative parallel methods [4, 5, 27, 28, 33, 34], first derivative parallel methods[3, 29, 30, 31, 36], second derivative parallel methods [8, 14] and parallel variable transformation algorithm [9].

Non-derivative parallel methods are based on the comparison of objective function values and there is no use of the derivatives. The construction of these methods has tow techniques, direct search and conjugate direction. Chazan and Miranker [4] describe a nongradient method similar in nature to Powell [22] and Zangwill [39] methods which requires no derivative computation and is a conjugate direction method. Powell and Zangwill algorithms as well as most unconstrained optimization algorithms usually proceed by a sequence of univariate minimizations. They proposed that this sequence can be performed simultaneously with degree of simultaneity as high as the dimension of the problem.

Sloboda [27, 28] proposed a method of conjugate directions for minimization not requiring the gradient information. He modified the projection method which used to solve linear algebraic systems by a suitable choice of an argument to become a new method of conjugate directions for minimize a strictly convex function. The method can be used on parallel computers where the minimizations on parallel directions are independent from the computational point of view and each processor of a multiprocessor system has to store only one vector [27]. In [32] Sutti has proposed a sequential conjugate direction algorithm for minimizing continuously differentiable strictly convex functions without using derivatives. The parallel algorithm has been given in [33, 34]. Dennis and Torczon [5] give a derivative–free approaches for unconstrained optimization that can be implement on parallel machines. The proposed algorithms are based on the simplex method [18]. Their experiments show that the speed–up is almost linear with the addition of more processors.

First derivative parallel methods assume that the gradient of the objective function is available. Quasi-Newton methods are effective methods for solving minimization problems. For finding a min-

imum of a function $f(x)$ using quasi-Newton methods we generate a sequence of points,

$$x_{k+1} = x_k - \alpha_k H_k g_k, \tag{1.2}$$

where $g_k = \nabla f(x_k)$, H_k is the inverse of the Hessian matrix of f evaluated at x_k, and α_k is a scalar chosen such that f is minimized along the direction $-H_k g_k$. These methods use only first derivatives to make an approximation H_k to the inverse of the Hessian matrix at each step instead of performing the computational work of evaluating and inverting Hessian matrix.

Straeter [31] suggested a parallel variable metric algorithm which is based on updating the inverse of the Hessian by a symmetric rank one update formula. Let ρ be the degree of parallelism, then the algorithm is defined as follows: the function and its gradient are computed in parallel at ρ different values of the independent variable; then the metric is modified by ρ rank-one corrections; and finally, a single univariant minimization is carried out in the Newton-like direction. He showed that the algorithm has a quadratic termination for any positive definite quadratic function. Van Laarhoven [36] classify Straeter's ideas [31] for parallel unconstrained optimization and apply them to Huang's class [12] of updating formulas. Huang's class contains Davidon-Fletcher-Powell (DFP) and Broyden-Fletcher-Goldfarb-Shanno (BFGS) formulas. He found that Straeter's rank-one updating formula appears to be the only parallel extension within Huang's class with the property of quadratic termination. Hence he developed a parallel extension of Broyden's rank-one updating formula [2] and prove quadratic termination.

In [3] the authors examine quasi-Newton methods for solving the unconstrained optimization problem on parallel computers. They try to parallelize both the function evaluation costs and the linear algebra calculations in the BFGS method. They also develop a class of new methods that fall in between the BFGS method and a finite difference Newton's method where the evaluation of the function, gradient, and part of the finite difference Hessian at each iteration is required. Still [29, 30] presented a parallel formulation of the BFGS method for unconstrained minimization by decomposing the iteration and update equations with an orthonormal basis. The numerical results indicate that the algorithm is effective in solving convex problems but are subject to the usual limitations of quasi-Newton methods. Bad initial approximations can cause the iterates to traverse infeasible regions or to diverge. To improve robustness he suggested to add a line search strategy which admits to parallel use and does not produce unnecessary function evaluations.

Second derivative parallel methods assume that the gradient and the Hessian of the objective function are available. In [8] a parallel Newton method is given for the minimization of a twice continuously differentiable uniformly convex function. The algorithm generates a sequence of points which converges superlinearly to a minimizer. The principle idea of the algorithm is to perform the calculation of the gradient vector, Hessian matrix, search direction and stepsize in parallel. They claim that the steps of the parallel algorithm never fail. This algorithm can be implemented in multiprocessor systems [23]. But it may never get the actual Hessian even it is positive definite. In [14] a parallel variant of the Newton method for unconstrained optimization, which uses as many finite differences of gradients as possible to update the inverse Hessian matrix is described. The method is based on the Gauss-Seidel type of updating for quasi–Newton methods which proposed by Straeter [31]. It incorporates the finite-difference approximations via the symmetric rank-one updates analysed by Van Laarhoven [36].

In [15] a parallel version called parallel gradient distribution is proposed. The parallel theorem allows each one of the parallel processors to use simultaneously a different algorithm, such as a descent, Newton, quasi-Newton, or conjugate gradient algorithm. Each processor can perform one or many steps of a serial algorithm on a portion of the gradient of the objective function assigned to it, independently of the other processors. At last a synchronization step is performed which, for

differentiable convex functions, consists of taking a strong convex combination of the points found by the processors. A more general synchronization step, applicable to convex as well as nonconvex functions, consists of taking the best point found by the processors or any point that is better. Multi-step, multi-directional parallel variable metric methods for unconstrained optimization are given in [21]. These algorithms generate variable metric directions at each iteration, different line search and scaling strategies are then applied in parallel along each search direction.

A general framework called parallel variable transformation algorithm is presented in [9]. The basic idea of the algorithm is to transform the variables into more than one space of smaller dimension simultaneously and compute candidate solutions on the latter spaces in parallel. The candidate solutions obtained are then used to generate an improved solution to the original problem. The parallel gradient distribution algorithm [15] is shown to be a special instances of the parallel variable transformation algorithm. A framework of the parallel variable transformation type algorithm, called the PVT-MYR algorithm, for minimizing a nonsmooth convex function is proposed in [19], which is constructed by converting an original objective function into a continuously differentiable function using the Moreau-Yosida regularization [16, 38]. A multithreaded parallel dual population genetic algorithm for unconstrained function optimizations on multi–core system is proposed in [35].

Secant methods are generally used on serial computers when the analytic Hessian matrix is unavailable or is too expensive to compute , and n is not too large. They use an approximation to the Hessian matrix that is updated by analyzing successive gradient vector and require n^2 storage and $\mathrm{O}(n^2)$ arithmetic operations per iteration [10].

In this paper, we concern with building quasi-Newton methods that are appropriate for parallel computers. We develop a parallel algorithm for a class of quasi-Newton methods for unconstrained minimization where the Hessian matrix is a function of a scalar parameter, all of which is positive definite and possess the quadratic convergence property. Also the DFP and BFGS matrices as well as Broyden family are special cases of this parametric family.

This paper is organized as follows: in Section 2, a description of a family of parallel quasi-Newton algorithms is introduced. The validity and reliability of the algorithm for solving unconstrained optimization problems on parallel computers is examined in Section 3. Finally, the paper ends with a conclusion in Section 4.

2. A family of parallel quasi-Newton algorithms

In this section we discuss the parallelization of a family of symmetric rank-two algorithms when it is applied to the solution of problem (1.1). The unconstrained optimization algorithms are iterative, which means that we can build a sequence of points that converges to a solution of the problem. They consisting of choosing the search direction which determining the rate of convergence of the algorithm and the stepsize which is important for the amount of calculations in an iteration.

2.1. The sequential method

Quasi-Newton methods maybe the widely used method for solving multivariate problem (1.1). It is suitable for problems where the number of variables is small enough that make the cost of storing an $n \times n$ matrix, and performing $\mathrm{O}(n^2)$ arithmetic operations per iteration, is acceptable; otherwise conjugate direction methods are the appropriate ones.

The iterative procedure of the used quasi-Newton method can be described as follows:

Algorithm 2.1.

Initialization

Start with an initial point $x_0 \in \mathbb{R}^n$, $g_0 = \nabla f(x_0)$ and a $n \times n$ positive definite symmetric matrix H_0 to approximate the inverse of the Hessian matrix of f. In the absence of additional information, H_0 is taken as the identity matrix I.

At iteration k

calculate search direction d_k

set $d_k = -H_k g_k$

calculate stepsize α_k

repeat

choose value of stepsize α_k

evaluate $f(x_k + \alpha_k d_k)$

until $x_k + \alpha_k d_k$ is acceptable

set $x_{k+1} = x_k + \alpha_k d_k$

evaluate $g_{k+1} = \nabla f(x_{k+1})$ if not already evaluated during line search

Test the new point x_{k+1} for optimality, if x_{k+1} is optimal terminate the iterative process, otherwise continue

update the inverse Hessian matrix approximation H_{k+1} [25]

$$H_{k+1} = H_k + \left(\left(\frac{t}{\sigma_k^T y_k} + \frac{(1-t)^2}{\left((1-t)\sigma_k - tH_k y_k\right)^T y_k} \right) \sigma_k - \frac{t(1-t)}{\left((1-t)\sigma_k - tH_k y_k\right)^T y_k} H_k y_k \right) \sigma_k^T$$
$$- \left(\frac{t(1-t)}{\left((1-t)\sigma_k - tH_k y_k\right)^T y_k} \sigma_k - \left(\frac{t^2}{\left((1-t)\sigma_k - tH_k y_k\right)^T y_k} - \frac{(1-t)}{y_k^T H_k y_k} \right) H_k y_k \right) \left(H_k y_k\right)^T \tag{2.1}$$

where $\sigma_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$.

The formula (2.1) represent a class of approximating matrices as a function of a scalar parameter t that includes the DFP and BFGS methods as special cases. When $t = 1$ gives DFP formula and

$$t = 1 + \frac{y_k^T B_k y_k + \sqrt{(y_k^T B_k y_k)^2 + 4\sigma_k^T y_k y_k^T B_k y_k}}{2\sigma_k^T y_k} \tag{2.2}$$

is the BFGS method. When

$$t = \frac{\sigma_k^T y_k + \theta(\sigma_k^T y_k + y_k^T H_k y_k) + \sqrt{\left(\sigma_k^T y_k + \theta(\sigma_k^T y_k + y_k^T H_k y_k)\right)^2 - 4\theta(\sigma_k^T y_k)^2}}{2\sigma_k^T y_k} \tag{2.3}$$

we can get the Broyden family, where $\theta \geq 0$. We can easily see that $t \geq 1$, if $\sigma_k^T y_k > 0$ and $y_k^T H_k y_k > 0$. The significance of the parameter t appear in the search direction that, it will not vanish and the algorithm stop when $g_{k+1} \neq 0$ and $t \geq 1$.

This description of the algorithm is appropriate for parallel, since it indicates to the important characteristics and costs of the method that can be needed in formulation of the parallel algorithm. There are two main sources of costs in Algorithm 2.1, function and derivative evaluations, and linear algebra calculations. Finding a suitable stepsize requires calculation of function evaluations at one or more trial points ending with the successful point x_{k+1}. Algorithm 2.1 has been tested on many problems, and we found that the successful point x_{k+1} is the first trial point and in some little cases more than one is needed. An average of $1.05 - 2.98$ trial points per iteration is found for many test problems. The gradient vector is calculated at the successful trial point during or after the search.

When the objective function is expensive to evaluate and no analytical procedure is available or it is difficult to calculate the gradient, the gradient at any point $\hat{x}$ is approximated by a finite difference formula. An approximation by the forward-difference approximation, can be given by

$$\frac{\partial f(\hat{x})}{\partial x_i} \simeq \frac{f(\hat{x} + \upsilon e_i) - f(\hat{x})}{\upsilon} \tag{2.4}$$

where e_i is the ith unit vector and υ is a small quantity. A more accurate approximation to the derivative can be obtained by using the central-difference formula, as

$$\frac{\partial f(\hat{x})}{\partial x_i} \simeq \frac{f(\hat{x} + \upsilon e_i) - f(\hat{x} - \upsilon e_i)}{2\upsilon} \tag{2.5}$$

The forward-difference approximation requires n evaluations of the objective function in addition to the evaluation at $\hat{x}$, and a total number $2n$ evaluations for the central-difference approximation. So when forward-difference gradients are used, each iteration of the Algorithm 2.1 usually requires $n+1$ to $n+3$ evaluations of $f(x)$, and a number $2n+1$ to $2n+3$ evaluations in case of central-difference approximations are used.

For the linear algebra computations in Algorithm 2.1, There are two main calculations, the calculation of the search direction d_k, and the calculation of the new inverse Hessian matrix approximation H_{k+1}. The calculation of search direction appears to require a small number of n^2 operations. The direction d_k is guaranteed to be a descent direction, because the inverse Hessian matrix approximation (2.1) is symmetric and positive definite. Also the calculation of H_{k+1} contains a rank two update formula that requires a small number of n^2 operations. Therefore the linear algebra costs are small for small n, and hence it is easy for function and derivative evaluation to be the dominant cost if gradients are replaced by finite difference approximations. But parallelism in linear algebra calculations of Algorithm 2.1 must be considered for the following reasons:

1. If these calculations are performed sequentially, they may reduce the performance on parallel computer.
2. There are some problems where number of variables are large and function evaluation cheap so that the linear algebra costs may be the dominant cost.

2.2. The parallel method

In most problems where $f(x)$ is expensive to evaluate, the gradient is not available analytically [3]. In this case it may be calculated by the finite difference approximation (2.4) or (2.5). The parallelism in an algorithm that uses finite difference gradients can be seen in the extra evaluations of $f(x)$ that can be executed concurrently. Equation (2.4) requires n/ρ concurrent function evaluation steps, and (2.5) involve $2n/\rho$ concurrent function evaluation steps where ρ is the number of processors. Determining a stepsize in line search uses only one processor and the remaining $\rho - 1$ processors are idle. Let τ be the extra evaluations of $f(x)$ that required for finite difference gradient calculations.

If ρ is small enough relative to τ that make $\rho/\tau \ll 1$, the idle processors do not effect on speedup since each gradient involves many concurrent function evaluation steps while a successful trial point requires 1 to 3 function evaluation steps.

If $\rho = \tau$ and parallelism occur only in the calculation of finite difference gradient, the maximum speedup that can be archived on problems with expensive function evaluation is at most half of optimal. The reason behind this is that the approximate gradient can be computed in one concurrent function evaluation step, and in line search a successful trial point requires at least one function evaluation. And there is $\rho - \tau$ processors are not used by this technique if $\rho > \tau$.

The idle processors in line search can be used according to the approach proposed by many authors such as [20, 36]. In that approach $\rho - 1$ processors that are available while evaluating $f(x)$ at $x_k + \alpha_k d_k$ can be used to evaluate $f(x)$ at other trial points in the search direction d_k and may also in other directions. but this technique changes the optimization algorithm and, sometimes gives in a better next iterate and thus a smaller total number of iterations being needed to solve the optimization problem [3].

Besides function and derivative evaluations, the dominant costs in Algorithm 2.1 at each iteration are the rank two update of the inverse H_k and the calculation of the search direction d_k. These calculation require $O(n^2)$ arithmetic operations and the other calculations in the algorithm require at most $O(n)$ operations. There are two cases for the update formula, Hessian update and inverse Hessian update. An utility of using the inverses update is that the calculation of the search directions becomes simple and cheap. The calculations of the update formula (2.1) can be organized as follows

$$
\begin{aligned}
& a = \sigma_k^T y_k, \quad z = H_k y_k, \\
& b = y_k^T z, \\
& c = (1-t)a - tb, \\
& \gamma = \frac{t(1-t)}{c}, \\
& \beta_1 = \frac{t}{a} + \frac{(1-t)^2}{c}, \quad \beta_2 = \frac{t^2}{c} - \frac{(1-t)}{b}, \\
& w_1 = \beta_1 \sigma_k - \gamma z, \quad w_2 = \gamma \sigma_k - \beta_2 z, \\
& H_{k+1} = H_k + w_1 \sigma_k^T - w_2 z. \qquad (2.6)
\end{aligned}
$$

Hence the dominant costs are one matrix vector multiplication, and a rank-two update formula of a symmetric matrix, are needed, each one requiring n^2 multiplications if the inverse Hessian is stored as lower or upper triangular matrix. On some parallel computes the full matrix is stored and hence the total cost of multiplications becomes $3n^2$.

The inverse updates appears more attractive for parallel computation but in optimization there is a belief that the unfactored inverse update may be less stable than the factored Hessian update. Authers in [3] have tested unfactored inverse update and the factored Hessian update experimentally of BFGS method on the test set of [17] and they found that the difference in performance were negligible, averaging no more than $1 - 2$ % overall with little variation on specific problems. Gradinetti [11] gets similar results. Therefore we use the inverse updates for the construction of parallel Algorithm 2.1.

The inverse updates seem to be suitable to implement on either shared or distributed memory multiprocessors. where only matrix-vector multiplications and rank-one updates are required, that

can be parallelize fully and implemented as block operations. The unfactored inverse approach should be implemented by storing the full inverse, which increases the number of arithmetic operations per processor to $3n^2/\rho$ but may reduce the memory access costs on a shared memory multiprocessor and to avoid excessive communication on a distributed memory multiprocessors [3].

The synchronization costs on a shared memory multiprocessor are small and the parallel algorithm should be efficient for almost any values of ρ. On a distributed memory multiprocessors, the communication costs per processor per iteration must be less than the costs of floating point operations for the processor in the iteration for the parallel algorithm to be efficient. Finally, Algorithm 2.1 contains $O(n)$ operations that can be parallelized to obtain a full parallel algorithm which is appropriate to apply on parallel computers.

3. Illustrative example

The main aim of this section is to illustrate the numerical performance of Algorithm 2.1 on test problems. We take the discrete integral equation function [17] as an example of test problems, which has the following form:

$$f_i(x) = \mathrm{x}_i + \frac{\zeta}{2}\left((1-\xi_i)\sum_{j=1}^{i}\xi_j(\mathrm{x}_j+\xi_j+1)^3 + \xi_i\sum_{j=i+1}^{n}(1-\xi_j)(\mathrm{x}_j+\xi_j+1)^3\right), \quad i=1,2,...,n \tag{3.1}$$

$$f(x) = \sum_{i=1}^{n} f_i^2(x), \tag{3.2}$$

$$\text{where} \quad x=(\mathrm{x}_1,\mathrm{x}_2,...,\mathrm{x}_n), \quad \zeta = \frac{1}{n+1}, \quad \xi_i = i\zeta. \tag{3.3}$$

The codes were written with Fortran 90 and OpenMP 3.1 in double precision arithmetic. All the tests were performed on a PC with AMD Phenom II x4 925 processor. The parallel performance of Algorithm 2.1 together with the sequential one are reported in Table 1 and Table 2. The algorithm is tested for both exact gradient calculations and central-difference approximations. The Armijo condition is used to determine the stepsize. The termination condition is $\| g_k \| \leq 10^{-4}$ where $\| \cdot \|$ denotes the Euclidean norm of vectors.
Tables 1 and 2 show the computation results, where the columns have the following meanings

Dim: the dimension of the test problem;

T: the total execution time of sequential algorithm in seconds;

T_ρ: the total execution time of parallel algorithm in seconds on ρ processors.

S_ρ: the speedup obtained with ρ processors.

E_ρ: the efficiency obtained with ρ processors.

From Table 1, we can see that parallel algorithm performance is acceptable. On one hand we see that the total time required by parallel algorithm is less than that of the sequential one, and the speedup obtained by $\rho = 2$ or $\rho = 3$ are greater than 1 for all tests. The maximum speedup obtained by $\rho = 2$ was 1.544 for $n = 9000$ while it was 1.727 for $\rho = 3$ and $n = 9000$. Also it can be seen that the speedup increases with the increment in number of variables n duo to the linear algebra calculations requires more computations and hence more time when n is increased. For $n = 100$

Table 1: Test results for Algorithm 2.1 with exact gradient

Dim	T	T_2	S_2	E_2	T_3	S_3	E_3
100	0.012	0.009	1.333	0.667	0.008	1.5	0.5
500	0.216	0.176	1.227	0.614	0.162	1.333	0.444
1000	0.829	0.667	1.243	0.622	0.607	1.366	0.455
2000	3.497	2.675	1.307	0.654	2.413	1.449	0.483
5000	22.231	17.203	1.292	0.646	15.383	1.445	0.482
7000	43.699	33.516	1.304	0.652	29.994	1.457	0.486
9000	87.475	56.662	1.544	0.772	50.639	1.727	0.576

Table 2: Test results for Algorithm 2.1 with approximate gradient

Dim	T	T_2	S_2	E_2	T_3	S_3	E_3
50	0.041	0.025	1.640	0.820	0.020	2.050	0.683
100	0.288	0.154	1.870	0.935	0.107	2.692	0.897
500	44.989	23.233	1.937	0.969	15.520	2.899	0.966
1000	347.671	178.981	1.943	0.972	118.934	2.923	0.974
2000	2860.381	1470.744	1.945	0.973	965.193	2.964	0.988

the speedup is greater than that for $n = 500, 1000, 2000, 5000, 7000$; because only one processor is evaluating the function $f(x)$ at the trial points $x_k + \alpha_k d_k$ during the line search and also evaluating the gradient vector at the successful points. For $n = 100$ the function and gradient evaluations are 8 and 6 respectively, while they was 10 and 8 for the others. The function and gradient evaluations of this function is expensive in time that reduce the speedup of the algorithm, which is also the reason behind that the speedup is not optimum.

On the other hand the maximum efficiency obtained by $\rho = 2$ is 0.772 and 0.576 for $\rho = 3$. This means that, on average, over the course of the execution, each of the processors is idle about 23% and 42% of the time for $\rho = 2$ and $\rho = 3$ respectively. A strategy that can be used to increase the speedup when the gradient evaluation has approximately the same execution time or less than that of the function evaluation is to evaluate the function and gradient concurrently during the line search which save the execution time for gradient calculations. If $x_k + \alpha_k d_k$ is not accepted as a successful point in line search, then this gradient information is not used by the algorithm, but nothing has been lost in execution time.

To evaluate the performance of Algorithm 2.1 when gradient approximations are used, we summarize the numerical results of the total execution time, speedup and efficiency for solving the discrete integral equation problem in Table 2. The approximated gradient calculations is expensive in function evaluations and can dominate on the calculation costs of Algorithm 2.1 for large number of variables. These calculations can be distributed between processors and executed in parallel. It can be seen from Table 2 that the speedup increases with the increment in number of variables n. The maximum speedup obtained by $\rho = 2$ was 1.945 while it was 2.964 for $\rho = 3$. This means that the speedup is optimal for the two cases and it increases with the increment in number of processors.

The maximum efficiency obtained by Algorithm 2.1 was 0.973 for $\rho = 2$ and 0.988 for $\rho = 3$. The obtained efficiency can be seen to be optimal and each of the processors is idle about 3% and 1% of the time for $\rho = 2$ and $\rho = 3$ respectively. It can be seem that Algorithm 2.1 is suitable for solving the unconstrained optimization problem (1.1) with exact gradient evaluations, but it is more suitable for solving problem (1.1) when gradient vector is not available analytically and finite difference gradient are used.

4. Conclusion

For finding solutions of unconstrained optimization problems on parallel computers, we have presented a parallel algorithm for a class of quasi-Newton methods. The linear algebra costs have been parallelize and can be utilize up to n processors since the linear algebra costs may be the dominant cost for large n and cheap function evaluation. An implementation of the presented algorithm to unconstrained optimization problem is given. If gradients are evaluated by finite differences, the function evaluations can be performed in parallel with efficiency achieved up to 98.8%. The tested problem indicate that the parallel algorithm are efficient and robust in solving large–scale problems. Finally, we observe that the proposed algorithm is much suitable for solving unconstrained optimization problems when gradient vector is not available analytically and finite difference gradient are used.

References

[1] A. I. Ahmed, *A new parameter free filled function for solving unconstrained global optimization problems*, Int. J. Comput. Math. 98 (2021) 106–119.

[2] C. G. Broyden, *A class of methods for solving nonlinear simultaneous equations*, Math. Comput. 19 (1965) 577–593.

[3] R. H. Byrd, R. B. Schnabel and G. A. Shultz, *Parallel quasi-Newton methods for solving the unconstrained optimization*, Math. Prog. 42 (1988) 273–306.

[4] D. Chazan and W. L. Miranker, *A nongradient and parallel algorithm for unconstrained minimization*, SIAM J. Cont. 8 (1970) 207–217.

[5] Jr J. E. Dennis and V. Torczon, *Direct search methods in parallel machines*, SIAM J. Opt. 1 (1991) 448–474.

[6] T. M. El–Gindy, M. S. Salim and A. I. Ahmed, *A Modified partial quadratic interpolation method for unconstrained optimization*, J. Conc. Appl. Math. 11 (2013) 136–146.

[7] T. M. El-Gindy, M. S. Salim and A. I. Ahmed, *A new filled function method applied to unconstrained global optimization*, Appl. Math. Comput. 273 (2016) 1246–1256.

[8] H. Fischer and K. Ritter, *An asynchronous parallel newton method*, Math. Prog. 42 (1988) 363–374

[9] M. Fukushima, *Parallel variable transformation in unconstrained optimization*, SIAM J. Opt. 8 (1998) 658–672.

[10] P. E. Gill, W. Murray and M. H. Wright, *Practical optimization*, Academic Press, London, 1981.

[11] L. Grandinetti, F*actorization versus nonfactorization in quasi-Newtonian methods for differentiable optimization*, Report N5, Dipartimento di Sistemi, Universita della Calabria 1978.

[12] H. Y. Huang, *Unified approach to quadratically convergent algorithms for function minimization*, J. Opt. Theo. Appl. 5 (1970) 405–423.

[13] J. K. Liu and S. J. Li, *New three–term conjugate gradient method with guaranteed global convergence*, Int. J. Comput. Math. 91 (2014) 1744–1754.

[14] F. A. Lootsma, *Parallel Newton–Raphson methods for unconstrained minimization with asynchronous updates of the Hessian matrix or its inverse*, in: M. Grauer, D. B. Pressmar (Eds.), Parallel computing and mathematical optimization,Springer-Verlag, Berlin, 1991, pp. 1-18.

[15] O. L. Mangasarian, *Parallel gradient distribution in unconstrained optimization*, SIAM J. Cont. Opt. 33 (1995) 1916–1925.

[16] J. Moreau, *Proximité et dualité dans un espace hilbertien*, Bull. Soc. Math. France, 93 (1965) 273–299.

[17] J. J. Moré, B. S. Garbow and K. E. Hillstrome, *Testing unconstrained optimization software*, ACM Trans. Math. Soft. 7 (1981) 17–41.

[18] J. A. Nelder and R. Mead, *A simplex method for function minimization*, Comput. J. 7 (1965) 308–313.

[19] L. P. Pang and Z. Q. Xia, *A PVT-TYPE algorithm for minimization a nonsmooth convex function*, Serdica Math. J. 29 (2003) 11–32.

[20] K. D. Patel, *Implementation of a parallel (SIMD) modified Newton method on the ICL DAP*, Technical Report No. 131, Numerical Optimisation Centre, The Hatfield Polytechnic (1982).

[21] P. K.-H. Phua, W. Fan, Y. Zeng, *Parallel algorithms for large-scale nonlinear optimization*, Int. Trans. Opl Res. 5 (1998) 67–77.

[22] M. J. D. Powell, *An efficient method for finding the minimum of a function of several variables without calculating derivatives*, Comput. J. 7 (1964) 155–162.

[23] K. Ritter, *Private communications in symposium on parallel optimization 3*, July 1993.
[24] M. S. Salim and A. I. Ahmed, *A piecewise polynomial approximation for solving nonlinear optimal control problems*, Far East J. Appl. Math. 95 (2016) 195–213.
[25] M. S. Salim and A. I. Ahmed, *A family of quasi-Newton methods for unconstrained optimization problems*, Opt. 67 (2018) 1717–1727.
[26] M. S. Salim and A. I. Ahmed, *A quasi-Newton augmented Lagrangian algorithm for constrained optimization problems*, J. Intel. Fuzzy Syst. 35 (2018) 2373–2382.
[27] F. Sloboda, *Parallel method of conjugate directions for minimization*, Apl. Mat. 20 (1975) 436–446.
[28] F. Sloboda, *A conjugate directions method and its applications*, Precedings of the 8th IFIP conference on Optimization Techniques 1977.
[29] C. H. Still, *Parallel quasi-Newton methods for unconstrained optimization*, Procedings of the fifth distributed memory computing conference, 263-271, April 8–12, 1990.
[30] C. H. Still, *The parallel BFGS Method for unconstrained minimization*, Procedings of the sixth distributed memory computing conference, 347-354, April 28–May 1 1991.
[31] T. A. Straeter, *A parallel variable metric optimization algorithm*, Technical report D-7329 NASA Langley Research center, Hampton, Va, 1973.
[32] C. Sutti, *A new method for unconstrained minimization without derivatives*, 277–289, In Towards global optimization, North Holland, 1978.
[33] C. Sutti, *Nongradient minimization methods for parallel processing computers*, Part 1, J. Opt. Theo. Appl. 39 (1983) 465-474.
[34] C. Sutti, *Nongradient minimization methods for parallel processing computers*, Part 2, J. Opt. Theo. Appl. 39 (1983) 475–488.
[35] A.J. Umbarkar, M.S. Joshi and W.-Ch. Hong, *Multithreaded parallel dual population genetic algorithm (MPDPGA) for unconstrained function optimizations on multi-core system*, Appl. Math. Comput. 243 (2014) 936–949.
[36] P. J. M. van Laarhoven, *Parallel variable metrix Algorithms for unconstrained optimization*, Math. Prog. 33 (1985) 68–81.
[37] Y. Xiao, Z. Wei and Z. Wang, *A limited memory BFGS–type method for large–scale unconstrained optimization*, Comput. Math. Appl. 56(4) (2008) 1001–1009.
[38] K. Yosida, *Functional analysis*, Springer Verlag, Berlin, 1964.
[39] W. I. Zangwill, *Minimizing a function without calculating derivatives*, Comput. J. 10 (1967) 293–296.