SQU Journal for Science, 2015, **20(2)**, 12-18

A Note on Using Partitioning Techniques for Solving Unconstrained Optimization Problems on Parallel Systems

Mehiddin Al-Baali[1] and Chefi Triki[2]*

[1] *Department of Mathematics and Statistics, Sultan Qaboos University, P.O. Box: 36, PC 123, Al-Khod, Muscat, Sultanate of Oman. Email: albaali@squ.edu.om.* [2]*Department of Mechanical and Industrial Engineering,Sultan Qaboos University, P.O. Box: 36, PC 123, Al-Khod, Muscat, Sultanate of Oman. *Email: chefi@squ.edu.om.*

ABSTRACT: We deal with the design of parallel algorithms by using variable partitioning techniques to solve nonlinear optimization problems. We propose an iterative solution method that is very efficient for separable functions, our scope being to discuss its performance for general functions. Experimental results on an illustrative example have suggested some useful modifications that, even though they improve the efficiency of our parallel method, leave some questions open for further investigation.

Keywords: Distributed systems; Parallel algorithms; Partitioning techniques; Unconstrained optimization.

ملاحظة حول استخدام تقنيات التجزئة لحل مسائل الأمثليات غير المقيدة على أنظمة متوازية

محي الدين البعلي و شافي التريكي

ملخص: نتعامل مع تصميم خوارزميات متوازية باستخدام متحول تقنيات التجزئة لحل مسائل الأمثليات غير الخطية. نقترح طريقة حل تكرارية تكون فعالة جدا لدوال منفصلة، ونطاق عملنا هو مناقشة أدائها على دوال عامة. بالاستناد إلى نتائج تجريبية على مثال توضيحي، تم اقتراح بعض التعديلات المفيدة. على الرغم أنها تحسن فعالية طريقتنا للتوازي، فإنها تترك بعض الأسئلة غير المحلولة لمتابعة البحث.

كلمات مفتاحية: الأمثليات غير المقيدة، الخوارزميات المتوازية، تقنيات التجزئة، أنظمة موزعة.

1. Introduction

The main aim of this work is to discuss the efficiency of solving nonlinear optimization problems on parallel systems when using partitioning techniques. Consider the following unconstrained optimization problem

$$\min_{x \in R^n} f(x) \tag{1}$$

that we would like to solve on parallel systems not by, trivially, parallelizing a given sequential method, but rather by designing algorithms specifically intended for parallel computers. We want to show that even though such techniques are very successful for some problems there is still the risk that the convergence of certain numerical methods happens very slowly for other types of problems.

Our approach consists of developing an algorithm that solves nonlinear optimization problems by partitioning the optimization problem into a given number of subproblems, each of which is solved separately by a sequential method. We try to achieve the partitioning phase in such a way that all processors are efficiently utilized in doing useful work during most of the execution time.

This strategy will be very fruitful when solving particular types of problems (e.g., separable functions) because, in this case, a solution of the original problem is obtained as soon as all subproblems are solved on the available processors of the parallel system. For general functions that are characterized by variable dependencies, however, convergence may be very slow, vanishing thus, any speedup deriving from the use of parallel technologies.

The solution of optimization problems on parallel systems or on grids has, over the past decades, attracted the attention of several researchers [1-4]. If we ignore those contributions that simply used the automatic parallelization offered by the intelligent compilers embedded within the parallel systems, then most of the works can be classified into two different types: fine and coarse grain parallelization. Fine grain parallelization acts directly on a matrix and vector

levels to split the workload among the processors [5, 6]. Coarse grain parallelization tries to tackle the structure of the problem in order to split it among the available processors [7-10]. This last direction was the one chosen by Ferris and Mangasarian, whose seminal paper [11] proposed a general paradigm for solving nonlinear optimization problems based on splitting the decision variables among the processors. Their method, called parallel variable distribution, involves, besides a concurrent phase, a synchronization phase based on calculating the affine hull of the generated partial solutions. Their method was proved theoretically to converge under a set of mathematical conditions and their preliminary experimental results showed the potential for highly efficient parallelization by the method.

After the seminal work of Ferris and Mangasarian many other methods have been proposed addressing parallelization of the problem's structure [12-15]. Our work here goes in this same direction to propose a parallel algorithm that distributes the variables of a general problem among the available processors. Then, unlike Ferris and Mangasarian's approach that uses an affine hull for the synchronization step, our iterative method simply restarts, whenever it is needed, the same method with a new initial point becoming available on some of the processors while running the sequential method independently.

The rest of our work is organized as follows. In section 2, we describe our proposed parallel algorithm. In section 3, we describe how to redefine the initial point for the parallel numerical algorithm. In section 4, we apply the proposed parallel algorithm to a simple but challenging general function. Then, in section 5 we suggest some modifications and discuss the numerical results. Finally, in section 6, we give our concluding remarks.

2. Parallel Algorithms for Unconstrained Optimization

We are interested in solving iteratively unconstrained optimization problems, as defined by (1), using a numerical method with a given initial point $x^{(1)}$.

For convenience, let the n variables of x be partitioned into groups that match the number p, say of processors of a parallel system. Specifically, let

$$x = (x^{[n_1]}, x^{[n_2]}, ..., x^{[n_i]}, ..., x^{[n_p]}),$$

where $n_1 + n_2 + ... + n_p = n.$ The initial point $x^{(1)}$ can then be defined as:

$$x^{(1)} = (x^{[n_1,1]}, x^{[n_2,1]}, ..., x^{[n_i,1]}, ..., x^{[n_p,1]}). \quad (2)$$

The idea of designing a parallel algorithm consists of solving iteratively, using the available processors, p independent unconstrained optimization subproblems starting from the initial point $x^{(1)}$. Then, in each iteration we check the termination condition,

$$\| g(x^{(c)}) \| \leq \varepsilon, \quad (3)$$

where $g(x)$ is the gradient of $f(x)$, ε is a tolerance parameter and c is an iteration counter. If condition (3) is satisfied, then the algorithm is terminated defining a proximate solution of (1). Otherwise, the process is restarted, but with a new initial point $x^{(c+1)}$, defined by using the solution obtained in iteration c, as will be described in Section 3.In the following, we refer to this repetitive process as *outer* iteration of the parallel method and use the index c for its identification, whereas the repetitive process inherent in the numerical method is referred to as *inner* iteration and denoted by the index k. Throughout the paper, we will specify explicitly the index we are referring to whenever this is not made evident from the context.

Consequently, in each iteration c, we apply on each processor p_i, $i = 1, 2, ..., p$, a certain sequential method [16, 17], to optimize subproblem:

$$\min_{y \in R^{n_i}} f_i(y) = \min_{y \in R^{n_i}} f_i(\bar{x}^{[n_i,c]}), \quad (4)$$

where

$$\bar{x}^{[n_i,c]} = (x^{[n_1,c]}, x^{[n_2,c]}, ..., x^{[n_{i-1},c]}, y, x^{[n_{i+1},c]}, ..., x^{[n_p,c]}). \quad (5)$$

Here all components of $\bar{x}$ are known quantities, except the partition y that represents the only set of variables of subproblem on p_i specified as

$$y = x^{[n_i]} = (x_{s+1}, x_{s+2}, ..., x_{s+n_i})^T,$$

where $s = n_0 + n_1 + n_2 + ... + n_{i-1}$ and $n_0 = 0$.

The starting point for a generic iteration c is, thus, given by

$$y^{(c)} = x^{[n_i,c]} = (x_{s+1}^{(c)}, x_{s+2}^{(c)}, ..., x_{s+n_i}^{(c)})^T.$$

After solving subproblem (4), we obtain on processor p_i a solution $y^* = x^{[n_i,*]}$ that verifies $f_i(y^*) = f_i(\bar{x}^{[n_i,*]})$. Then by combining all the results deriving from all the processors, a new estimate for a solution of problem (1) is defined.

It is worth noting that this algorithm is very efficient if the variables assigned to each processor p_i are independent of the other variables, i.e., the original problem can be partitioned into a set of independent subproblems. In this case, the solution of the original problem (1) is obtained as soon as all the processors complete the required tasks of solving the assigned subproblems. This happens, for example, in the case of separable problems in which it is possible to group the variables to distribute among a certain number of processors as equally as possible, each requiring almost the same time for solving its assigned subproblem. Subsequently, by combining all the obtained partial solutions we get the desired solution of (1). In this case, the time required to solve (1) is equal to the time required to solve the most difficult subproblem plus some data communication overhead.

However, for general problems, it is usual to expect that the variables assigned to processor p_i depend on some other variables handled by another processor p_j with $i \neq j$. In this case, the algorithm may converge very slowly, as shown in Section 4, since it will be necessary to restart the algorithm by generating a new initial point at each (outer) iteration of our method. The way to generate a new initial solution is illustrated in the next section, and some improvements will be proposed in Section 5.

3. Obtaining a New Starting Point

We assume here that, at a given iteration c, not all the processors terminated with a solution $\bar{x}^{[n_i,c]}$ of the assigned subproblem i which, once combined with the other partial results, does not constitute a valid solution to the original problem (i.e., condition (3) is not satisfied). In this case, we may exploit any solution $\bar{x}^{[n_j,c]}$ of another subproblem that is available on processor, say p_j, and defined by (5) with i replaced by j, that is $\bar{x}^{[n_j,c]}$.

The idea is to make this value available to processor p_i, which will restart the process of solving problem (4) with $x^{(c)}$ as defined above, except that $\bar{x}^{[n_j,c]}$ is replaced by $\bar{x}^{[n_j,*]}$. In addition, in order to improve the quality of the initial solution, it is also advantageous to replace $\bar{x}^{[n_j,c]}$ by $\bar{x}^{[n_i,c]}$, whenever we have

$$f_i(\bar{x}^{[n_j,c]}) < f_i(\bar{x}^{[n_i,c]}).$$

4. Illustrative Example

Consider solving the (general) well-known Rosenbrock problem

$$\min_{x \in R^2} f(x) = 100(x_2 - x_1^2)^2 + (1-x_1)^2, \tag{6}$$

starting with $x^{(1)} = (0,0)^T$, as in [18]. It is easy to note that the optimal solution is $x^* = (1,1)^T$.

Assume that two processors, p_1 and p_2, are available. In this case, $n_1 = n_2 = 1$ and $x^{(1)} = (x^{[1,1]}, x^{[2,1]})^T = (0,0)^T$, but for simplicity of notation we will rename the two variables as $x^{(1)} = (y^{(1)}, z^{(1)})^T = (0,0)^T$.

Consequently, the corresponding subproblem (4) to be solved on p_1 can be written as

$$\min_{y \in R} f_1(y) = 100(z - y^2)^2 + (1-y)^2, \tag{7}$$

where $z = x^{[2,1]}$ is considered to be a known constant here (set initially to 0) and the starting point, for $c = 1$, is given by $y^{(c)} = 0$.

Similarly, processor p_2 should solve the subproblem

$$\min_{z \in R} f_2(z) = 100(z - y^2)^2 + (1-y)^2, \tag{8}$$

where $y = x^{[1,1]}$ is a known constant (set initially to 0) and the initial point, for $c = 1$, is given by $z^{(c)} = 0$.

While problem (8) can be solved on p_2 in one step exactly , easily yielding the solution $z^* = y^2$, we need to apply a numerical method to solve subproblem (7). Here, we have implemented the Newton-Raphson method, which is known to be very efficient whenever the starting point is close enough to the optimal point [19]. By using the index k to denote the inner iterations of this iterative method, its termination criterion can then be defined by

$$\| y^{(k)} - y^{(k+1)} \| \leq \varepsilon, \tag{9}$$

where $\varepsilon = 10^{-6}$.

Numerical results for solving (7) on processor p_1 are reported in Table 1. For each inner iteration k, it includes the value of the variable $y^{(k)}$, its corresponding value of $f_1(y^{(k)})$ and the initial value of $z^{(c)}$ corresponding to the (outer) iteration c (recall that $z^{(c)}$ is constant for problem f_1).

Table 1. Solution of problem (7) on p_1 for outer iteration $c = 1$.

k	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	0	1	0
2	1	100	0
3	6.6722130×10^{-1}	19.929645	0
4	4.4688493×10^{-1}	4.294191	0
5	3.0373406×10^{-1}	1.3358733	0
6	2.1664151×10^{-1}	8.3392632×10^{-1}	0
7	1.7376816×10^{-1}	7.7383512×10^{-1}	0
8	1.6209454×10^{-1}	7.7112126×10^{-1}	0
9	1.6126604×10^{-1}	7.7110970×10^{-1}	0
10	1.6126202×10^{-1}	7.7110970×10^{-1}	0

From the results of Table 1, it is clear that the Newton-Raphson method has terminated because of the satisfaction of condition (9). However, the whole algorithm is not terminated because condition (3) is not satisfied yet. For this reason, we need to restart a new outer iteration by using a new estimate of the initial solution $x^{(2)} = (0.16126202, 0)^T$ as obtained from the solution of problem (1) as reported in Table 1 (and from the exact solution of subproblem (8)). The results as summarized in Table 2 show that condition (3) is not verified yet, and thus the iterative process should continue by using $x^{(3)} = (0.2113389, 0.02600544)^T$ as a new initial solution.

Table 2. Solution of problem (7) on p_1 for outer iteration $c = 2$.

k	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	1.6126202×10^{-1}	7.0348137×10^{-1}	0
2	2.3482151×10^{-1}	6.7038702×10^{-1}	0
3	2.1393910×10^{-1}	6.5695530×10^{-1}	0
4	2.1137604×10^{-1}	6.5680104×10^{-1}	0
5	2.1133890×10^{-1}	6.5680104×10^{-1}	0

Before proceeding, we want to note that the total number of function evaluations, say *NF*, required to terminate the Newton-Raphson method is 10 in Table 1 and 5 in Table 2. This tradition of counting the number of function evaluations as a measure of the computing cost of a method is very important in nonlinear optimization. In fact it represents an alternative way (with respect to the CPU run time) of measuring the performance of a numerical algorithm without relating it to a specific hardware platform. For this reason, we will even use this measure to define a termination condition of the whole parallel algorithm. We decide for simplicityto stop our parallel algorithm on the basis of the following well-known condition, instead of using criterion (3):

$$\| f_1(y^{(c)}) - f_1(y^{(c+1)}) \| \leq \eta, \tag{10}$$

where c is the outer iterations index and η is chosen to be, for example, 10^{-8}.

A summary of the results of using condition (10) in the solution of our illustrative example is reported in Table 3. This table, besides reporting k, $y^{(k)}$, $f_1(y^{(k)})$ and $z^{(c)}$ as before, also shows the number of outer iterations, c, and the number of function evaluations, *NF*. The first two lines of Table 3 can be easily derived from the results of Tables 1 and 2 and the value of $x^{(3)}$, as defined above, can be seen in the line corresponding to $c = 3$.

Table 3. Outer and inner iterations for solving problem (7) on p_1.

c	k	NF	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
0	0	-	0	1	0
1	10	10	1.6126202×10^{-1}	7.7110970×10^{-1}	0
2	5	15	2.1133890×10^{-1}	6.5680104×10^{-1}	2.6005440×10^{-2}
3	4	19	2.4508301×10^{-1}	5.9362041×10^{-1}	4.4664126×10^{-2}
4	4	22	2.7112272×10^{-1}	5.4933041×10^{-1}	6.0065686×10^{-2}
:	:	:	:	:	:
:	:	:	:	:	:
10	4	47	3.6653980×10^{-1}	4.0873867×10^{-1}	1.2571033×10^{-1}
11	3	50	3.7761509×10^{-1}	3.9415559×10^{-1}	1.3435143×10^{-1}
:	:	:	:	:	:
:	:	:	:	:	:
208	3	641	8.0786871×10^{-1}	3.7055832×10^{-2}	6.5146273×10^{-1}
209	2	643	8.0860185×10^{-1}	3.6773699×10^{-2}	6.5265184×10^{-1}
:	:	:	:	:	:
:	:	:	:	:	:
1892	2	4009	9.9800324×10^{-1}	3.9970463×10^{-6}	9.9600046×10^{-1}
1893	2	4011	9.9800825×10^{-1}	3.9770370×10^{-6}	9.9601048×10^{-1}

Even though our parallel algorithm succeeds in generating the solution $x^* = (0.99800825, 0.99601048)^T$ which is close to the optimal point $(1,1)^T$, these results show that its performance, expressed in terms of the number of function evaluations, remains unsatisfactory. A positive comment that can be drawn from the results is that as the number of outer iterations increases, the quality of the initial solution improves and, consequently, the number of inner iterations required to solve f_1 decreases.

5. Modified Algorithm

The example in the previous section highlighted the main drawback of solving nonseparable problems on parallel systems, namely the fact that some of the processors remain idle while others are doing useful work. In the previous example, indeed, processor p_1 was heavily utilized whereas p_2 solved its task in one step and remained idle most of the time. In order to rectify this inefficiency, we propose a modification of our algorithm consisting of calling upon any useful detail made available on any of the processors to update the initial solution and restart the process. We illustrate this idea by reconsidering the example of Section 4.

Suppose that processor p_2, once idle, calls upon any detail made so far available on p_1, i.e. $y^{(k=2)} = 1$(see Table 1). Although $f_1(y^{(k=2)}) = 100$ is much larger than $f_1(y^{(k=1)}) = 1$, this will result in getting $z^{(2)} = 1*1 = 1$ on p_2 and, surprisingly, in immediately obtaining the optimal solution of the original problem. Since this case may happen rarely in practice, we propose another practical way of modifying our parallel method.

While solving subproblem (7) on p_1, we do not wait until the satisfaction of condition (9), but we stop the inner iterations earlier, i.e., whenever we note

$$f_1(y^{(k)}) < f_1(y^{(k=1)}). \tag{11}$$

We then pass this newly generated value to the other processor to be used as a new initial value. The whole process will then terminate only when condition (10) holds. To understand how this modification affects the solution of our example, consider the results of Table 1. By applying the new rule (11), processor p_1 does not need to run 10 inner iterations anymore, but stops at iteration 6, i.e., as soon as $f_1(y^{(k=6)}) = 0.83392632$, which turns to be less than $f_1(y^{(k=1)}) = 1$, thus reducing the time in which p_2 remains idle. Hence, both processors terminate with the point $x^{(2)} = (0.21664151, 0)^T$. Using this new estimate, we apply this procedure again in order to solve problem (8) and continue until obtaining the final estimate of the solution of (8) which yields that of (6). The results obtained are summarized in Table 4.

Comparing these results with those reported in Table 3, we note how our modification substantially reduced the number of inner iterations k required in each outer iteration and, consequently, the number of function evaluations (NF) is reduced from 4011 to 1894.

Table 4. Results of the modified algorithm for solving problem (7) on p_1.

c	k	NF	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	6	6	2.1664151×10^{-1}	8.3392632×10^{-1}	0
2	1	7	2.5625824×10^{-1}	5.8825087×10^{-1}	4.6933546×10^{-2}
3	1	8	2.8353419×10^{-1}	5.3500097×10^{-1}	6.5668290×10^{-2}
4	1	9	3.0514270×10^{-1}	4.9900761×10^{-1}	8.0391645×10^{-2}
:	:	:	:	:	:
:	:	:	:	:	:
10	1	14	3.8824373×10^{-1}	3.8099867×10^{-1}	1.4251556×10^{-1}
11	1	15	3.9822456×10^{-1}	3.6829531×10^{-1}	1.5073320×10^{-1}
:	:	:	:	:	:
:	:	:	:	:	:
208	1	212	8.1014949×10^{-1}	3.6180865×10^{-2}	6.5516895×10^{-1}
209	1	213	8.1086987×10^{-1}	3.5906568×10^{-2}	6.5634220×10^{-1}
:	:	:	:	:	:
:	:	:	:	:	:
1889	1	1893	9.9800360×10^{-1}	3.9956158×10^{-6}	9.9600118×10^{-1}
1890	1	1894	9.9800860×10^{-1}	3.9756095×10^{-6}	9.9601119×10^{-1}

Although this last modification has improved our parallel algorithm significantly, the high value of NF required to solve the problem remains unsatisfactory. The modified method seems to converge, but slowly indeed, since a large NF, albeit smaller than before, is still required. A further reduction in the number of outer iterations (and thus in NF) can be observed only when larger values of η are used. In this case, however, the substantial reductions of the values of c and NF (that can be observed in Table 5 for increasing values of η) are counterbalanced by a deterioration in the quality of the solution obtained for problem (6). It is the responsibility of the decision maker to define the acceptable tradeoff between the solution's quality and the method's efficiency.

Table 5. Results of the modified algorithm for different values of η.

η	c	k	NF	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
10^{-8}	1890	1	1894	9.9800860×10^{-1}	3.9756095×10^{-6}	9.9601119×10^{-1}
10^{-6}	989	1	994	9.8044276×10^{-1}	3.8347995×10^{-4}	9.6116828×10^{-1}
10^{-4}	244	1	249	8.3378845×10^{-1}	2.7725833×10^{-2}	6.9420540×10^{-1}
10^{-3}	59	1	64	6.1549985×10^{-1}	1.4882825×10^{-1}	3.7569698×10^{-1}

6. Concluding Remarks

We proposed a parallel algorithm based on a variable partitioning technique designed to efficiently solve separable problems, aiming to test its usefulness for general problems. We observed that, despite the proof of convergence demonstrated in [11], such a convergence may happen very slowly, probably vanishing the speedup deriving from the use of parallel systems. The main cause of this disadvantage is the fact that some of the processors remain idle while others are engaged with carrying out useful computations. We proposed some modifications that succeeded in reducing the effect of such inefficiency, but the results still turned out to be unsatisfactory. The solution of a simple (but not separable) function required a large number of function evaluations, while the use of the sequential Newton method is known to require only 3 function evaluations [16]. On the basis of the above observations, the following points are made:

- while our proposed method is efficient for separable problems, it converges slowly for general functions;
- the illustrative example, a general function of a small size, faced the difficulty of having one of the two processors remaing idle during most of the computation time. This showed our original algorithm to be inefficient for solving nonseparable functions;
- for well-structured general functions allowing a balanced distribution of the workload among the available processors, we expect better performance from our algorithm. Moreover, tackling large scale problems will increase the workload, and thus reduce the effect of unavoidable waiting time of the idle processors;
- the improvements that can be achieved by our method are clearly problem dependent and the remarkable speedup obtained in solving our illustrative example may prove to be significant for other problems. Thinking about more general improvements tailored for nonseparable functions (such as considering load balancing techniques [20] or tasks scheduling [21]) may be of great benefit to speed up our parallel method.

Carrying out more intensive experiments and developing general purpose improvement techniques are beyond the scope of this paper. This contribution wanted, on the one side, to highlight the challenges faced when solving general problems by using partitioning techniques on parallel systems and, on the other side, to raise new problems of concern for further investigation.

References

1. Schnabel, R.B. A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Computing* . 1995, **21(6),** 875-905.
2. Bertsekas, D.P. and Tsitsiklis, J.N. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, Belmont-USA, 1997.
3. Triki, C. and Grandinetti L. Computational grids to solve large scale optimization problems with uncertain data. *International Journal of Computing* 2002, **1(1),** 20-26.
4. Migdalas, A., Toraldo, G. and Kumar, V. Nonlinear optimization and parallel computing. *Parallel Computing* 2003, **29(4),** 375-391.
5. Beraldi, P., Grandinetti, L., Musmanno, R. and Triki, C. Parallel algorithms to solve two-stage stochastic linear programs with robustness constraints. *Parallel Computing*. 2000, **26(13-14),** 1889-1908.
6. D'Apuzzo, M. and Marino, M. Parallel computational issues of an interior point method for solving large bound-constrained quadratic programming problems. *Parallel Computing* 2003, **29(4),** 467-483.
7. Blomvall. J. A multistage stochastic programming algorithm suitable for parallel computing. *Parallel Computing*, 2003, **29(4),** 431-445.
8. Giraud, L., Haidar, A. and Pralet. S. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing*, 2010, **36(5-6),** 285-296.
9. Argello, F., Heras, D.B., Bo, M. and Lamas-Rodrguez, J. The split-and-merge method in general purpose computation on GPUs. *Parallel Computing*, 2012, **38(6-7),** 277-288.
10. Triki, C. Solving the flood propagation problem with Newton algorithm on parallel systems. *SQU Journal for Science*, 2012, **17(1),** 147-156,
11. Ferris, M.C. and Mangasarian, O.L. Parallel variable distribution. *SIAM Journal on Optimization*, 1994, **4,** 815-832.
12. Mangasarian, O.L. Parallel gradient distribution in unconstrained optimization. *SIAM Journal on Optimization*, 1995, **33,** 1916-1925.
13. Rotiroti, D., Triki, C. and Grandinetti, L. Combined MPI/OpenMP implementations for a stochastic programming solver. In *Parallel Computing Advances and Current Issues* (Edited by G. Joubert, A. Murli, F. Peters and M. Vanneschi), Imperial College Press, 2002.
14. Zeng. D. Parallel iterative methods for nonlinear programming problems. *Advanced Materials Research*, 2010, **159,**105-110.
15. Li, W. A parallel multi-start search algorithm for dynamic traveling salesman problem. *Lecture Notes in Computer Science*, 2011, **6630,** 65-75.
16. Fletcher, R. *Practical Methods of Optimization.* Wiley, New York, 1987.
17. Al-Baali, M. and Khalfan, H. An overview of some practical quasi-Newton methods for unconstrained optimization. *SQU Journal for Science*, 2007, **12(2),** 199-209.
18. Al-Baali, M. Solving Optimization Problems on Parallel Computers. Technical Report DMCS 1/95, UAE University, UAE, 1995
19. Ruszczynski, A.P. *Nonlinear Optimization.* Princeton University Press, New Jersey, 2006
20. Grama, A. and Kumar, V. Load balancing for parallel optimization techniques. Encyclopedia of Optimization, 2008, 1905-1911.
21. Sinnen, O. *Task Scheduling for Parallel Systems.* Wiley-Interscience, New Jersey, 2007

Received 19 June 2014
Accepted 7 September 2014